IEEE

IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

and the

IEEE Standards Association Corporate Advisory Group

1800™

IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language

Abstract: This standard provides a set of extensions to the IEEE 1364™ Verilog® hardware description language (HDL) to aid in the creation and verification of abstract architectural level models. It also includes design specification methods, embedded assertions language, testbench language including coverage and an assertions application programming interface (API), and a direct programming interface (DPI). This standard enables a productivity boost in design and validation and covers design, simulation, validation, and formal assertion-based verification flows.
Keywords: assertions, design automation, design verification, hardware description language, HDL, PLI, programming language interface, SystemVerilog, Verilog, Verilog programming interface, VPI

Introduction

This introduction is not a part of IEEE Std 1800-2005, IEEE Standard for SystemVerilog: Unified Hardware Design, Specification, and Verification Language.

The purpose of this standard is to provide the electronic design automation (EDA), semiconductor, and system design communities with a well-defined and official IEEE unified hardware design, specification, and verification standard language. The language is designed to coexist and enhance the hardware description languages (HDLs) presently used by designers while providing the capabilities lacking in those languages.

SystemVerilog is a unified hardware design, specification, and verification language that is based on the Accellera SystemVerilog 3.1a extensions to the Verilog HDL [B1][a], published in 2004. Accellera is a consortium of EDA, semiconductor, and system companies. IEEE Std 1800 enables a productivity boost in design and validation and covers design, simulation, validation, and formal assertion-based verification flows.

SystemVerilog enables the use of a unified language for abstract and detailed specification of the design, specification of assertions, coverage, and testbench verification that is based on manual or automatic methodologies. SystemVerilog offers application programming interfaces (APIs) for coverage and assertions, a vendor-independent API to access proprietary waveform file formats, and a direct programming interface (DPI) to access proprietary functionality. SystemVerilog offers methods that allow designers to continue to use present design languages when necessary to leverage existing designs and intellectual property. This standardization project will provide the VLSI design engineers with a well-defined IEEE standard that meets their requirements in design and validation and enables a step function increase in their productivity. This standardization project will also provide the EDA industry with a standard to which they can adhere and which they can support in order to deliver their solutions in this area.

Notice to users

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http://standards.ieee.org/reading/ieee/updates/errata/index.html. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: http://standards.ieee.org/reading/ieee/interp/index.html.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

---

[a]The numbers in brackets correspond to the numbers in the bibliography in Annex K.

Participants

The SystemVerilog Language Working Group is entity based. At the time this standard was completed, the SystemVerilog Working Group had the following membership:

Johny Srouji, IBM, *Chair*
Shrenik Mehta, Sun Microsystems, Inc., *Vice Chair*
Dennis Brophy, Mentor Graphics Corporation, *Secretary*
Karen Pieper, Synopsys, Inc., *Technical Chair*
Stuart Sutherland, Sutherland HDL, Inc., *Technical Editor*

Victor Berman, Cadence Design Systems, Inc.
Clifford Cummings, Sunburst Design, Inc.
Vassilios Gerousis, Accellera
Oz Levia, Synopsys, Inc.

Yossi Levy, Intel Corporation
Rishiyur Nikhil, Bluespec, Inc.
Alec Stanculescu, Fintronic USA

Work on this standard was divided among five primary committees.

The Champions Committee was responsible for ensuring consistency in the work done by each committee, as well as between this standard and the Verilog standard (IEEE Std 1364[b]).

Karen Pieper, Synopsys, Inc., *Chair*
Tom Fitzpatrick, Mentor Graphics Corporation, *Co-Chair*

Shalom Bresticker, Intel Corporation
Surrendra Dudani, Synopsys, Inc.
Neil Korpusik, Sun Microsystems, Inc.
Francoise Martinolle, Cadence Design Systems, Inc.

Brad Pierce, Synopsys, Inc.
Dave Rich, Mentor Graphics Corporation
Stuart Sutherland, Sutherland HDL, Inc.
Bassam Tabbara, Novas Software, Inc.

The Basic/Design Committee (SV-BC) was responsible for the specification of the design features of SystemVerilog.

Matt Maidment, Intel Corporation, *Chair*
Brad Pierce, Synopsys, Inc., *Co-Chair*

Cliff Cummings, Sunburst Design, Inc.
Surrendra Dudani, Synopsys, Inc.
Mark Hartoog, Synopsys, Inc.
Dan Jacobi, Intel Corporation
Greg Jaxon, Synopsys, Inc.
Francoise Martinolle, Cadence Design Systems, Inc.
Kathy McKinley, Cadence Design Systems, Inc.
Don Mills, LCDM Engineering

Rishiyur Nikhil, Bluespec, Inc.
Karen Pieper, Synopsys, Inc.
Logie Ramachandran, Synopsys, Inc.
Dave Rich, Mentor Graphics Corporation
Steven Sharp, Cadence Design Systems, Inc.
Stuart Sutherland, Sutherland HDL, Inc.
Gordon Vreugdenhil, Mentor Graphics Corporation
Doug Warmke, Mentor Graphics Corporation

The Enhancement Committee (SV-EC) was responsible for the specification of the testbench features of SystemVerilog.

Mehdi Mohtashemi, Synopsys, Inc., *Chair*
Neil Korpusik, Sun Microsystems, Inc., *Co-Chair*

Stefen Boyd, Boyd Technology, Inc.
Dennis Brophy, Mentor Graphics Corporation
Michael Burns, Motorola, Inc.
Kevin Cameron, Sonics, Inc.
Cliff Cummings, Sunburst Design, Inc.
Surrendra Dudani, Synopsys, Inc.
Jeff Freedman, Mentor Graphics Corporation

Jay Lawrence, Cadence Design Systems, Inc.
Francoise Martinolle, Cadence Design Systems, Inc.
Brad Pierce, Synopsys, Inc.
Dave Rich, Mentor Graphics Corporation
Ray Ryan, Mentor Graphics Corporation
Arturo Salz, Synopsys, Inc.
Stuart Sutherland, Sutherland HDL, Inc.

---

[b]Information on references can be found in Clause 2.

The Assertions Committee (SV-AC) was responsible for the specification of the assertion features of SystemVerilog.

Faisal Haque, Cisco Systems, Inc., *Chair*
Arif Samad, Synopsys, Inc., *Co-Chair*

Doron Bustan, Freescale Semiconductor, Inc.
Ed Cerny, Synopsys, Inc.
Surrendra Dudani, Synopsys, Inc.
John Havlicek, Motorola, Inc.

Manisha Kulshrestha, Mentor Graphics Corporation
Joseph Lu, NVIDIA Corporation
Hillel Miller, Freescale Semiconductor, Inc.
Bassam Tabbara, Novas Software, Inc.

The C API Committee (SV-CC) was responsible for on the specification of the DPI, the assertions and coverage APIs, and the Verilog programming interface (VPI) features of SystemVerilog.

Charles Dawson, Cadence Design Systems, Inc., *Chair*
Ghassan Khoory, Synopsys, Inc., *Co-Chair*

Tapati Basu, Synopsys, Inc.
Steven Dovich, Cadence Design Systems, Inc.
Ralph Duncan, Mentor Graphics Corporation
Jim Garnett, Mentor Graphics Corporation
Joao Geada, CLK Design Automation
Andrzej Litwiniuk, Synopsys, Inc.
Francoise Martinolle, Cadence Design Systems, Inc.
Sachchidananda Patel, Synopsys, Inc.

Michael Rohleder, Motorola, Inc.
Rob Slater, Freescale Semiconductor, Inc.
John Stickley, Mentor Graphics Corporation
Stuart Sutherland, Sutherland HDL, Inc.
Bassam Tabbara, Novas Software, Inc.
Jim Vellenga, Cadence Design Systems, Inc.
Doug Warmke, Mentor Graphics Corporation

The following members of the entity balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Accellera
Bluespec, Inc.
Cadence Design Systems, Inc.
Fintronic USA
IBM
Infineon Technologies
Intel Corporation

JEITA
Mentor Graphics Corporation
Sun Microsystems, Inc.
Sunburst Design, Inc.
Sutherland HDL, Inc.
Synopsys, Inc.
Synplicity

When the IEEE-SA Standards Board approved this standard on 8 November 2005, it had the following membership:

Steve M. Mills, *Chair*
Richard H. Hulett, *Vice Chair*
Don Wright, *Past Chair*
Judith Gorman, *Secretary*

Mark D. Bowman
Dennis B. Brophy
Joseph Bruder
Richard Cox
Bob Davis
Julian Forster*
Joanna N. Guenin
Mark S. Halpin
Raymond Hapeman

William B. Hopf
Lowell G. Johnson
Herman Koch
Joseph L. Koepfinger*
David J. Law
Daleep C. Mohla
Paul Nikolich

T. W. Olsen
Glenn Parsons
Ronald C. Petersen
Gary S. Robinson
Frank Stone
Malcolm V. Thaden
Richard L. Townsend
Joe D. Watson
Howard L. Wolfman

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Satish K. Aggarwal, *NRC Representative*
Richard DeBlasio, *DOE Representative*
Alan H. Cookson, *NIST Representative*

Michelle D. Turner
IEEE Standards Project Editor

Contents

IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language

1. Overview

1.1 Scope

This standard specifies extensions for a higher level of abstraction for modeling and verification with the Verilog® hardware description language (HDL). These additions extend Verilog into the systems space and the verification space. SystemVerilog is built on top of IEEE Std 1364™[1] for the Verilog HDL. This standard includes design specification methods, embedded assertions language, testbench language including coverage and assertions application programming interface (API), and a direct programming interface (DPI).

Throughout this standard, the following terms apply:

— *Verilog* refers to IEEE Std 1364 for the Verilog HDL.

— *Verilog-2001* refers to IEEE Std 1364-2001 [B4][2] for the Verilog HDL.

— *Verilog-1995* refers to IEEE Std 1364-1995 [B3] for the Verilog HDL.

— *SystemVerilog* refers to the extensions to the Verilog standard (IEEE Std 1364) as defined in this standard.

1.2 Purpose

SystemVerilog is built on top of IEEE Std 1364. SystemVerilog improves the productivity, readability, and reusability of Verilog-based code. The language enhancements in SystemVerilog provide more concise hardware descriptions, while still providing an easy route with existing tools into current hardware implementation flows. The enhancements also provide extensive support for directed and constrained-random testbench development, coverage-driven verification, and assertion-based verification.

---

[1]Information on references can be found in Clause 2.
[2]The numbers in brackets correspond to the numbers in the bibliography in Annex K.

SystemVerilog adds extended and new constructs to Verilog, including the following:

— Extensions to data types for better encapsulation and compactness of code and for tighter specification

  — C data types: **int**, **typedef**, **struct**, **union**, **enum**

  — Other data types: bounded queues, logic (0, 1, X, Z) and bit (0, 1), tagged unions for safety

  — Dynamic data types: string, classes, dynamic queues, dynamic arrays, associative arrays including automatic memory management freeing users from deallocation issues

  — Dynamic casting and bit-stream casting

  — Automatic/static specification on a per-variable-instance basis

— Extended operators for concise description

  — Wild equality and inequality

  — Built-in methods to extend the language

  — Operator overloading

  — Streaming operators

  — Set membership

— Extended procedural statements

  — Pattern matching on selection statements for use with tagged unions

  — Enhanced loop statements plus the foreach statement

  — C-like jump statements: return, break, continue

  — **final** blocks that execute at the end of simulation (inverse of **initial**)

  — Extended event control and sequence events

— Enhanced process control

  — Extensions to **always** blocks to include synthesis consistent simulation semantics

  — Extensions to **fork…join** to model pipelines and for enhanced process control

  — Fine-grain process control

— Enhanced tasks and functions

  — C-like void functions

  — Pass by reference

  — Default arguments

  — Argument binding by name

  — Optional arguments

  — Import/export functions for DPI

— Classes: object-oriented mechanism that provides abstraction, encapsulation, and safe pointer capabilities

— Automated testbench support with random constraints

— Interprocess communication synchronization

  — Semaphores

  — Mailboxes

  — Event extensions, event variables, and event sequencing

— Clarification and extension of the scheduling semantics

— Cycle-based functionality: **clocking** blocks and cycle-based attributes that help reduce development, ease maintainability, and promote reusability

  — Cycle-based signal drives and samples

  — Synchronous samples

— Race-free program context
— Assertion mechanism for verifying design intent and functional coverage intent
    — Property and sequence declarations
    — Assertions and coverage statements with action blocks
— Extended hierarchy support
    — Packages for declaration encapsulation with import for controlled access
    — Compilation-unit scope nested modules and extern modules for separate compilation support
    — Extension of port declarations to support interfaces, events, and variables
    — $root to provide unambiguous access using hierarchical references
— Interfaces to encapsulate communication and facilitate communication-oriented design
— Functional coverage
— DPI for clean, efficient interoperation with other languages (C provided)
— Assertion API
— Coverage API
— Data read API
— Verilog procedural interface (VPI) extensions for SystemVerilog constructs
— Concurrent assertion formal semantics

1.3 Conventions used in this standard

This standard is organized into clauses, each of which focuses on a specific area of the language. There are subclauses within each clause to discuss individual constructs and concepts. The discussion begins with an introduction and an optional rationale for the construct or the concept, followed by syntax and semantic descriptions, followed by examples and notes.

The terminology conventions used throughout this standard are as follows:
— The word *shall* is used to indicate mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*).
— The word *should* is used to indicate that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended that*).
— The word *may* is used to indicate a course of action permissible within the limits of the standard (*may* equals *is permitted to*).
— The word *can* is used for statements of possibility and capability, whether material, physical, or causal (*can* equals *is able to*).

1.4 Syntactic description

The main text uses the following conventions:
— *Italicized* font when a term is being defined
— `Constant-width` font for examples, file names, and references to constants, especially `0`, `1`, `x`, and `z` values
— **`Boldface constant-width`** font for Verilog and SystemVerilog keywords, when referring to the actual keyword

The formal syntax of SystemVerilog is described using Backus-Naur Form (BNF). The following conventions are used:

— Lowercase words, some containing embedded underscores, denote syntactic categories. For example:

module_declaration

— **Boldface-red** characters denote reserved keywords, operators, and punctuation marks as a required part of the syntax. For example:

module   =>   ;

— A vertical bar (|) separates alternative items, unless it appears in boldface-red, in which case it stands for itself. For example:

unary_operator ::=
+ | **-** | **!** | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^** | **^~**

— Square brackets ([ ]) enclose optional items. For example:

input_declaration ::= **input [** range **]** list_of_variables **;**

— Braces ({ }) enclose a repeated item, unless it appears in boldface-red, in which case it stands for itself. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent:

list_of_param_assignments ::= param_assignment { **,** param_assignment }

list_of_param_assignments ::=
    param_assignment
  | list_of_param_assignment **,** param_assignment

— If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, "*msb*_index" and "*lsb*_index" are equivalent to "index."

1.5 Use of color in this standard

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in pure black and white. The places where color is used are the following:

— Cross references that are hyperlinked to other portions of this standard are shown in underlined-blue text (hyperlinking works when this standard is viewed interactively as a PDF file).

— Syntactic keywords and tokens in the formal language definitions are shown in **boldface-red text**.

— Some figures use a minimal amount of color to enhance readability.

1.6 Contents of this standard

A synopsis of the clauses and annexes is presented as a quick reference. There are 31 clauses and 11 annexes. All clauses, as well as Annex A through Annex I, are normative parts of this standard. Annex J and Annex K are included for informative purposes only.

Clause 1 describes the contents of this standard and the conventions used in this standard.

Clause 2 lists references to other standards that are required in order to implement this standard.

Clause 3 describes the lexical tokens used in SystemVerilog source text and their conventions. It describes how to specify and interpret the lexical tokens.

Clause 4 describes enhancements to Verilog data objects and data types, including new variable data objects and types, net data type extensions, string types, enumerated types, user-defined types, structures, and unions.

Clause 5 describes SystemVerilog arrays, including packed and unpacked arrays, dynamic arrays, associative arrays, queues, and various array methods.

Clause 6 describes declaring net, variable and constant data, enhanced rules on writing to variables, signal aliasing, and type compatibility.

Clause 7 describes the object-oriented programming capabilities in SystemVerilog. Topics include defining classes, dynamically constructing objects, inheritance and subclasses, data hiding and encapsulation, polymorphism, and parameterized classes.

Clause 8 describes new operators, rules on operations with SystemVerilog data types, operations on arrays, operator methods, and operator overloading.

Clause 9 describes SystemVerilog enhanced simulation scheduling semantics.

Clause 10 describes enhancements to Verilog decision statements and looping constructs, new procedural statements, `final` blocks, statement and block labels, enhanced event types and event control.

Clause 11 describes specialized procedural blocks for modeling combinational logic, latched logic, and sequential logic; enhancements to Verilog continuous assignments; and process control.

Clause 12 describes numerous enhancements to Verilog tasks and functions, plus the syntax for importing functions from a foreign language and exporting tasks and functions to a foreign language.

Clause 13 describes generating random numbers, constraining random number generation, dynamically changing constraints, and seeding random number generators (RNGs) and randomized `case` statement execution.

Clause 14 describes built-in semaphore and mailbox classes and describes enhanced Verilog event type and operators.

Clause 15 defines `clocking` blocks, input and output skews, cycle delays, and default clocking.

Clause 16 describes the testbench program construct, the elimination of testbench race conditions, and program control tasks.

Clause 17 describes immediate assertions, concurrent assertions, properties, sequences, sequence operations, multiclock sequences, clock resolution, and assertion binding.

Clause 18 describes coverage groups, coverage points, cross coverage, coverage options, and coverage methods.

Clause 19 describes packages, compilation units ($unit), top-level instance ($root), nested modules, extern modules, enhanced port declarations, time unit and precision, port connection rules, and name spaces.

Clause 20 describes interface syntax, interface ports, modports, interface tasks and functions, parameterized interfaces, virtual interfaces, and accessing objects within interfaces.

Clause 21 describes enhancements to Verilog configurations.

Clause 22 describes several extensions to Verilog system tasks and system functions, including a $typename function, $bits size function, range function, array querying functions, assertion control tasks, random number functions, program control tasks, coverage functions, and enhancements to Verilog system tasks and system functions.

Clause 23 describes extensions to the Verilog `define and `include directives, and a new directives for controlling keywords compatibility.

Clause 24 describes extensions to the value change dump (VCD) file to support SystemVerilog data objects and data types.

Clause 25 covers the possible deprecation of the Verilog `defparam` statement and the Verilog procedural `assign`/`deassign` statements.

Clause 26 describes SystemVerilog's direct interface to foreign languages.

Clause 27 describes enhancements to the VPI object diagrams to support SystemVerilog constructs.

Clause 28 describes the assertion API in SystemVerilog.

Clause 29 describes the coverage API in SystemVerilog.

Clause 30 describes enhancements to the VPI to support accessing design and simulation data from files.

Annex A defines the formal syntax of SystemVerilog, using BNF.

Annex B lists the SystemVerilog keywords.

Annex C describes the system type definitions for mailbox, semaphore, randomize, and process.

Annex D defines a List package that implements a list data-structure, analogous to the standard template library (STL).

Annex E describes a formal semantics for SystemVerilog concurrent assertions.

Annex F defines the C-language layer for the SystemVerilog DPI.

Annex G defines the standard `svdpi.h` include file for use with SystemVerilog DPI applications.

Annex H describes common guidelines for the inclusion of foreign language code into a SystemVerilog application.

Annex I provides a listing of the contents of the `sv_vpi_user.h` file, which extends the Verilog `vpi_user.h` include file.

Annex J defines terms that are used in this standard.

Annex K lists reference documents that are related to this standard.

1.7 Examples

Several small SystemVerilog code examples are shown throughout this standard. These examples are informative. They are intended to illustrate the usage of SystemVerilog constructs in a simple context and do not define the full syntax.

1.8 Prerequisites

This standard presupposes a working knowledge of the Verilog HDL and the Verilog programming language interface (PLI). Some clauses of this standard presuppose a working knowledge of the C programming language.

2. Normative references

The following referenced documents are indispensable for the application of this standard. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1364™, IEEE Standard for Verilog Hardware Description Language.[3, 4, 5]

IEEE Std 754™, IEEE Standard for Binary Floating-Point Arithmetic.

---

[3]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

[4]This IEEE standards project was not approved by the IEEE-SA Standards Board at the time this publication went to press. For information about obtaining a draft, contact the IEEE.

[5]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

3. Literal values

3.1 Introduction

NOTE—The lexical conventions for SystemVerilog literal values are extensions of those for Verilog. SystemVerilog adds literal time values, literal array values, literal structures, and enhancements to literal strings.[6]

3.2 Literal value syntax

```
time_literal[5] ::=                                                          // from A.8.4
        unsigned_number time_unit
      | fixed_point_number time_unit
time_unit ::= s | ms | us | ns | ps | fs | step
number ::=                                                                   // from A.8.7
        integral_number
      | real_number
integral_number ::=
        decimal_number
      | octal_number
      | binary_number
      | hex_number
decimal_number ::=
        unsigned_number
      | [ size ] decimal_base  unsigned_number
      | [ size ] decimal_base  x_digit { _ }
      | [ size ] decimal_base  z_digit { _ }
binary_number ::= [ size ] binary_base  binary_value
octal_number ::= [ size ] octal_base  octal_value
hex_number ::= [ size ] hex_base  hex_value
sign ::= + | -
size ::= non_zero_unsigned_number
non_zero_unsigned_number[1] ::= non_zero_decimal_digit { _ | decimal_digit}
real_number[1] ::=
        fixed_point_number
      | unsigned_number [ . unsigned_number ] exp [ sign ] unsigned_number
fixed_point_number[1] ::= unsigned_number . unsigned_number
exp ::= e | E
unsigned_number[1] ::= decimal_digit { _ | decimal_digit }
string_literal ::= " { Any_ASCII_Characters } "                              // from A.8.8
```

Syntax 3-1—Literal values (excerpt from Annex A)

---

[6]Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement this standard.

3.3 Integer and logic literals

Literal integer and logic values can be sized or unsized, and they follow the same rules for signedness, truncation, and left-extending as Verilog.

SystemVerilog adds the ability to specify unsized literal single-bit values with a preceding apostrophe ('), but without the base specifier. All bits of the unsized value are set to the value of the specified bit. In a self-determined context, these literals have a width of 1 bit, and the value is treated as unsigned.

```
'0, '1, 'X, 'x, 'Z, 'z    // sets all bits to this value
```

3.4 Real literals

The default type is **real** for fixed-point format (e.g., `1.2`), and exponent format (e.g., `2.0e10`).

A cast can be used to convert literal **real** values to the **shortreal** type (e.g., `shortreal'(1.2)`). Casting is described in [4.14](#).

3.5 Time literals

Time is written in integer or fixed-point format, followed without a space by a time unit (**fs ps ns us ms s step**). For example:

```
0.1ns
40ps
```

The time literal is interpreted as a **realtime** value scaled to the current time unit and rounded to the current time precision.

NOTE—While s, ms, ns, ps, and fs are the usual SI unit symbols for second, millisecond, nanosecond, picosecond, and femtosecond, due to lack of the Greek letter µ (mu) in coding character sets, 'us' represents the SI unit symbol for microsecond, properly µs.

3.6 String literals

A string literal is enclosed in quotes and has its own data type. Nonprinting and other special characters are preceded with a backslash. SystemVerilog adds the following special string characters:

```
\v vertical tab
\f form feed
\a bell
\x02 hex number
```

A string literal must be contained in a single line unless the new line is immediately preceded by a \ (back slash). In this case, the back slash and the new line are ignored. There is no predefined limit to the length of a string literal.

A string literal can be assigned to an integral type, as in Verilog. If the size differs, it is right justified.

```
byte c1 = "A" ; bit [7:0] d = "\n" ;
bit [0:11] [7:0] c2 = "hello world\n" ;
```

A string literal can be assigned to an unpacked array of bytes. If the size differs, it is left justified.

```
byte c3 [0:12] = "hello world\n" ;
```

Packed and unpacked arrays are discussed in Clause 5. The difference between string literals and array literals is discussed in 3.7.

String literals can also be cast to a packed or unpacked array, which shall follow the same rules as assigning a literal string to a packed or unpacked array. Casting is discussed in 4.14.

SystemVerilog also includes a `string` data type to which a string literal can be assigned. Variables of type string have arbitrary length; they are dynamically resized to hold any string. String literals are packed arrays (of a width that is a multiple of 8 bits), and they are implicitly converted to the string type when assigned to a string type or used in an expression involving string type operands (see 4.7).

3.7 Array literals

Array literals are syntactically similar to C initializers, but with the replicate operator ({{}}) allowed.

```
int n[1:2][1:3] = '{'{0,1,2},'{3{4}}};
```

The nesting of braces must follow the number of dimensions, unlike in C. However, replicate operators can be nested. The inner pair of braces in a replication is removed. A replication expression only operates within one dimension.

```
int n[1:2][1:6] = '{2{'{3{4, 5}}}}; // same as '{'{4,5,4,5,4,5},'{4,5,4,5,4,5}}
```

Array literals are array assignment patterns or pattern expressions with constant member expressions (see 8.13.1). An array literal must have a type, which may be either explicitly indicated with a prefix or implicitly indicated by an assignment-like context.

```
typedef int triple [1:3];
$mydisplay(triple'{0,1,2});
```

Array literals can also use their index or type as a key and use a default key value (see 8.13.1).

```
triple b = '{1:1, default:0};  // indexes 2 and 3 assigned 0
```

3.8 Structure literals

Structure literals are structure assignment patterns or pattern expressions with constant member expressions (see 8.13.2). A structure literal must have a type, which may be either explicitly indicated with a prefix or implicitly indicated by an assignment-like context.

```
typedef struct {int a; shortreal b;} ab;
ab c;
c = '{0, 0.0}; // structure literal type determined from
               // the left-hand context (c)
```

Nested braces should reflect the structure. For example:

```
ab abarr[1:0] = '{'{1, 1.0}, '{2, 2.0}};
```

The C-like alternative '{1, 1.0, 2, 2.0} for the preceding example is not allowed.

Structure literals can also use member name and value or use data type and default value (see [8.13.2](#)):

```
c = '{a:0, b:0.0};                // member name and value for that member
c = '{default:0};                 // all elements of structure c are set to 0
d = ab'{int:1, shortreal:1.0};    // data type and default value for all members
                                  // of that type
```

When an array of structures is initialized, the nested braces should reflect the array and the structure. For example:

```
ab abarr[1:0] = '{'{1, 1.0}, '{2, 2.0}};
```

Replicate operators can be used to set the values for the exact number of members. The inner pair of braces in a replication is removed.

```
struct {int X,Y,Z;} XYZ = '{3{1}};
typedef struct {int a,b[4];} ab_t;
int a,b,c;
ab_t v1[1:0] [2:0];
v1 = '{2{'{3{a,'{2{b,c}}}}}};

    /* expands to '{'{3{'{a,{2{b,c}}}}}, '{3{{a,'{2{b,c}}}}}} */

    /* expands to '{'{'{a,'{2{b,c}}},'{a,'{2{b,c}}},'{a,'{2{b,c}}}},
                    '{'{a,'{2{b,c}}},'{a,'{2{b,c}}},'{a,'{2{b,c}}} } } */

    /* expands to '{'{'{a,'{b,c,b,c}},'{a,'{b,c,b,c}},'{a,'{b,c,b,c}}},
                    '{'{a,'{b,c,b,c}},'{a,'{b,c,b,c}},'{a,'{b,c,b,c}}}} */
```

4. Data types

4.1 Introduction

NOTE—To provide for clear translation to and from C, SystemVerilog supports the C built-in types, with the meaning given by the implementation C compiler. However, to avoid the duplication of **int** and **long** without causing confusion, **int** is 32 bits and **longint** is 64 bits in SystemVerilog. The C **float** type is called **shortreal** in SystemVerilog so that it is not be confused with the Verilog **real** type.

Verilog has data objects that can take on values from a small number of predefined value systems: the set of 4-state logic values, vectors and arrays of logic values, and the set of floating point values. SystemVerilog extends Verilog by introducing some of the data types that conventional programming languages provide, such as enumerations and structures.

In extending the type system, SystemVerilog makes a distinction between an object and its data type. A data type is a set of values and a set of operations that can be performed on those values. Data types can be used to declare data objects or to define user-defined data types that are constructed from other data types.

The Verilog logic system is based on a set of four state values: 0, 1, X, and Z. Although this 4-state logic is fundamental to the language, it does not have a name. SystemVerilog has given this primitive data type a name, **logic**. This new name can be used to declare objects and to construct other data types from the 4-state data type.

The additional strength information associated with bits of a net is not considered part of the data type.

SystemVerilog adds **string**, **chandle**, and **class** data types and enhances the Verilog **event** type.

Verilog provides arbitrary fixed-length arithmetic using 4-state logic. The 4-state type can have bits at X or Z, however. Therefore, it may be less efficient than an array of bits because the operator evaluation must check for X and Z, and twice as much data must be stored. SystemVerilog adds a **bit** data type that can only have bits with 0 or 1 values. See 4.3.2 on 2-state data types.

User-defined types are introduced by **typedef** and must be defined before they are used. Data types can also be parameters to modules or interfaces, making them like class templates in object-oriented programming. One routine can be written to reverse the order of elements in any array. Such a routine is impossible in C and in Verilog.

Structures and unions are complicated in C because the tags have a separate name space. SystemVerilog follows the C syntax, but without the optional structure tags.

See also Clause 5 on arrays.

4.2 Data type syntax

```
data_type ::=                                                                    // from A.2.2.1
        integer_vector_type [ signing ] { packed_dimension }
      | integer_atom_type [ signing ]
      | non_integer_type
      | struct_union [ packed [ signing ] ] { struct_union_member { struct_union_member } }
            { packed_dimension }13
      | enum [ enum_base_type ] { enum_name_declaration { , enum_name_declaration } }
      | string
      | chandle
      | virtual [ interface ] interface_identifier
      | [ class_scope | package_scope ] type_identifier { packed_dimension }
      | class_type
      | event
      | ps_covergroup_identifier
      | type_reference28
enum_base_type ::=
        integer_atom_type [ signing ]
      | integer_vector_type [ signing ] [ packed_dimension ]
      | type_identifier [ packed_dimension ]24
enum_name_declaration ::=
        enum_identifier [ [ integral_number [ : integral_number ] ] ] [ = constant_expression ]
class_scope ::= class_type ::
class_type ::=
        ps_class_identifier [ parameter_value_assignment ]
            { :: class_identifier [ parameter_value_assignment ] }
integer_type ::= integer_vector_type | integer_atom_type
integer_atom_type ::= byte | shortint | int | longint | integer | time
integer_vector_type ::= bit | logic | reg
non_integer_type ::= shortreal | real | realtime
net_type ::= supply0 | supply1 | tri | triand | trior | trireg | tri0 | tri1 | uwire | wire | wand | wor
signing ::= signed | unsigned
simple_type ::= integer_type | non_integer_type | ps_type_identifier
struct_union_member26 ::=
        { attribute_instance } [random_qualifier] data_type_or_void  list_of_variable_decl_assignments ;
data_type_or_void ::= data_type | void
struct_union ::= struct | union [ tagged ]
type_reference ::=
        type ( expression27 )
      | type ( data_type )
variable_decl_assignment ::=                                                     // from A.2.4
        variable_identifier { variable_dimension } [ = expression ]
      | dynamic_array_variable_identifier [ ] [ = dynamic_array_new ]
      | class_variable_identifier [ = class_new ]
      | [ covergroup_variable_identifier ] = new [ ( list_of_arguments ) ]16
```

Syntax 4-1—Data types (excerpt from Annex A)

4.3 Integer data types

SystemVerilog offers several integer data types, representing a hybrid of both Verilog and C data types, as shown in Table 4-1.

Table 4-1—Integer data types

`shortint`	2-state SystemVerilog data type, 16-bit signed integer
`int`	2-state SystemVerilog data type, 32-bit signed integer
`longint`	2-state SystemVerilog data type, 64-bit signed integer
`byte`	2-state SystemVerilog data type, 8-bit signed integer or ASCII character
`bit`	2-state SystemVerilog data type, user-defined vector size
`logic`	4-state SystemVerilog data type, user-defined vector size
`reg`	4-state Verilog data type, user-defined vector size
`integer`	4-state Verilog data type, 32-bit signed integer
`time`	4-state Verilog data type, 64-bit unsigned integer

4.3.1 Integral types

The term *integral* is used throughout this standard to refer to the data types that can represent a single basic integer data type, `packed array`, `packed struct`, `packed union`, `enum`, or `time`.

The term *simple bit vector type* is used throughout this standard to refer to the data types that can directly represent a one-dimensional packed array of bits. The packed vector types of Verilog are simple bit vector types, as are the integral types with predefined widths, such as byte. The SystemVerilog packed structure types and multidimensional packed array types are not simple bit vector types, but each is equivalent (see 6.9.2) to some simple bit vector type, to and from which it can be easily converted.

4.3.2 2-state (two-value) and 4-state (four-value) data types

Types that can have unknown and high-impedance values are called *4-state types*. These are `logic`, `reg`, `integer`, and `time`. The other types do not have unknown values and are called *2-state types*, for example, `bit` and `int`.

The difference between `int` and `integer` is that `int` is a 2-state type and `integer` is a 4-state type. The 4-state values have additional bits that encode the X and Z states. The 2-state data types can simulate faster, take less memory, and are preferred in some design styles.

The Verilog keyword `reg` does not always accurately describe user intent. SystemVerilog adds the keyword `logic` as a more descriptive term. The keywords `logic` and `reg` are equivalent types (see 6.9.2 for details on type equivalence).

Automatic type conversions from a smaller number of bits to a larger number of bits involve zero extensions if unsigned or sign extensions if signed. Automatic type conversions from a larger number of bits to a smaller number of bits involve truncations of the most significant bits (MSBs). When a 4-state value is automatically converted to a 2-state value, any unknown or high-impedance bits shall be converted to zeros.

4.3.3 Signed and unsigned data types

Integer types use integer arithmetic and can be signed or unsigned. This affects the meaning of certain operators such as '<', etc.

```
int unsigned ui;
int signed si;
```

The data types **byte**, **shortint**, **int**, **integer**, and **longint** default to **signed**. The data types **bit**, **reg**, and **logic** default to **unsigned**, as do arrays of these types.

The **signed** keyword in the preceding example is part of Verilog. The **unsigned** keyword is a reserved keyword in Verilog, but is not utilized.

See also Clause 8 on operators and expressions.

4.4 Real and shortreal data types

The **real**[7] data type is from Verilog and is the same as a C **double**. The **shortreal** data type is a SystemVerilog data type and is the same as a C **float**.

4.5 Void data type

The **void** data type represents nonexistent data. This type can be specified as the return type of functions to indicate no return value. This type can also be used for members of tagged unions (see 4.11).

4.6 Chandle data type

The **chandle** data type represents storage for pointers passed using the DPI (see Clause 26). The size of a value of this data type is platform dependent, but shall be at least large enough to hold a pointer on the machine in which the tool is running.

The syntax to declare a handle is as follows:

```
chandle variable_name ;
```

where `variable_name` is a valid identifier. Chandles shall always be initialized to the value **null**, which has a value of 0 on the C side. Chandles are very restricted in their usage, with the only legal uses being as follows:

— Only the following operators are valid on **chandle** variables:
  — Equality (**==**), inequality (**!=**) with another **chandle** or with **null**
  — Case equality (**===**), case inequality (**!==**) with another **chandle** or with **null** (same semantics as **==** and **!=**)
— Chandles can be tested for a boolean value that shall be 0 if the **chandle** is **null** and 1 otherwise.
— Only the following assignments can be made to a **chandle**:
  — Assignment from another **chandle**
  — Assignment to **null**

---

[7]The **real** and **shortreal** types are represented as described by IEEE Std 754.

— Chandles can be inserted into associative arrays (refer to <u>5.9</u>), but the relative ordering of any two entries in such an associative array can vary, even between successive runs of the same tool.

— Chandles can be used within a class.

— Chandles can be passed as arguments to functions or tasks.

— Chandles can be returned from functions.

The use of chandles is restricted as follows:

— Ports shall not have the **chandle** data type.

— Chandles shall not be assigned to variables of any other type.

— Chandles shall not be used as follows:

— In any expression other than as permitted in this subclause

— As ports

— In sensitivity lists or event expressions

— In continuous assignments

— In untagged unions

— In packed types

4.7 String data type

SystemVerilog includes a **string** data type, which is an ordered collection of characters. The length of a **string** variable is the number of characters in the collection. Variables of type **string** are dynamic as their length may vary during simulation. A single character of a **string** variable may be selected for reading or writing by indexing the variable. A single character of a **string** variable is of type **byte**. SystemVerilog also includes a number of special methods to work with strings.

Verilog supports string literals, but only at the lexical level. In Verilog, string literals behave like packed arrays of a width that is a multiple of 8 bits. A string literal assigned to a packed array of an integral variable of a different size is either truncated to the size of the variable or padded with zeroes to the left as necessary.

In SystemVerilog, string literals behave exactly the same as in Verilog. However, SystemVerilog also supports the **string** data type to which a string literal can be assigned. When using the **string** data type instead of an integral variable, strings can be of arbitrary length and no truncation occurs. Literal strings are implicitly converted to the **string** type when assigned to a **string** type or used in an expression involving **string** type operands.

The indices of **string** variables shall be numbered from 0 to N–1 (where N is the length of the string) so that index 0 corresponds to the first (leftmost) character of the string and index N–1 corresponds to the last (rightmost) character of the string. The **string** variables can take on the special value "", which is the empty string. Indexing an empty string variable shall be an out-of-bounds access.

A **string** shall not contain the special character `"\0"`. Assigning the value 0 to a **string** character shall be ignored.

The syntax to declare a **string** is as follows:

```
string variable_name [= initial_value];
```

where `variable_name` is a valid identifier and the optional `initial_value` can be a string literal or the value "" for an empty string. For example:

```
    string myName = "John Smith";
```

If an initial value is not specified in the declaration, the variable is initialized to "", the empty string. An empty string has zero length.

SystemVerilog provides a set of operators that can be used to manipulate combinations of string variables and string literals. The basic operators defined on the **string** data type are listed in Table 4-2.

A string literal can be assigned to a **string** or an integral type. If their size differs, the literal is right justified and either truncated on the left or zero filled on the left, as necessary. For example:

```
    byte c = "A";              // assign to c "A"
    bit [10:0] a = "\x41";     // assigns to a 'b000_0100_0001
    bit [1:4][7:0] h = "hello" ;  // assigns to h "ello"
```

A **string** or a string literal can be assigned directly to a **string** variable. Integral types can be assigned to a **string** variable, but require a cast. When casting an integral value to a string, the **string** variable shall grow or shrink to accommodate the integral value. If the size (in bits) of the integral value is not a multiple of 8, then the integral value is zero filled on the left.

A string literal assigned to a **string** variable is converted according to the following steps:
— All "\0" characters in the string literal are ignored (i.e., removed from the **string**).
— If the result of the first step is an empty string literal, the **string** is assigned the empty string.
— Otherwise, the **string** is assigned the remaining characters in the string literal.

Casting an integral value to a **string** variable proceeds in the following steps:
— If the size (in bits) of the integral value is not a multiple of 8, the integral value is left extended and filled with zeros until its bit size is a multiple of 8. The extended value is then treated the same as a string literal, where each successive 8 bits represent a character.
— The steps described above for string literal conversion are then applied to the extended value.

For example:

```
    string s1 = "hello";       // sets s1 to "hello"
    bit [11:0] b = 12'ha41;
    string s2 = string'(b);    // sets s2 to 16'h0a41
```

As a second example:

```
    typedef reg [15:0] r_t;
    r_t r;
    integer i = 1;
    string b = "";
    string a = {"Hi", b};

    r = r_t'(a);        // OK
    b = string'(r);     // OK
    b = "Hi";           // OK
    b = {5{"Hi"}};      // OK
    a = {i{"Hi"}};      // OK (non constant replication)
    r = {i{"Hi"}};      // invalid (non constant replication)
    a = {i{b}};         // OK
    a = {a,b};          // OK
    a = {"Hi",b};       // OK
    r = {"H",""};       // yields "H\0". "" is converted to 8'b0
```

```
b = {"H",""};          // yields "H". "" is the empty string
a[0] = "h";            // OK, same as a[0] = "cough"
a[0] = b;              // invalid, requires a cast
a[1] = "\0";           // ignored, a is unchanged
```

Table 4-2—String operators

Operator	Semantics
Str1 == Str2	*Equality.* Checks whether the two strings are equal. Result is 1 if they are equal and 0 if they are not. Both strings can be of type **string**. Or one of them can be a string literal. If both operands are string literals, the operator is the same Verilog equality operator as for integer types.
Str1 != Str2	*Inequality.* Logical negation of ==
Str1 < Str2 Str1 <= Str2 Str1 > Str2 Str1 >= Str2	*Comparison.* Relational operators return 1 if the corresponding condition is true using the lexicographical ordering of the two strings Str1 and Str2. The comparison uses the compare string method. Both operands can be of type **string**, or one of them can be a string literal.
{Str1,Str2,...,Strn}	*Concatenation.* Each operand can be of type **string** or a string literal (it shall be implicitly converted to type **string**). If at least one operand is of type **string**, then the expression evaluates to the concatenated string and is of type **string**. If all the operands are string literals, then the expression behaves like a Verilog concatenation of integral types; if the result is then used in an expression involving **string** types, it is implicitly converted to the **string** type.
{multiplier{Str}}	*Replication.* Str can be of type **string** or a string literal. Multiplier must be of integral type and can be nonconstant. If multiplier is nonconstant or **Str** is of type **string**, the result is a string containing *N* concatenated copies of Str, where N is specified by the multiplier. If Str is a literal and the multiplier is constant, the expression behaves like numeric replication in Verilog (if the result is used in another expression involving string types, it is implicitly converted to the string type).
Str[index]	*Indexing.* Returns a byte, the ASCII code at the given index. Indexes range from 0 to *N*–1, where *N* is the number of characters in the string. If given an index out of range, returns 0. Semantically equivalent to Str.getc(index), in 4.7.3.
Str.method(...)	The dot (.) operator is used to invoke a specified method on strings.

SystemVerilog also includes a number of special methods to work with strings. These methods use the built-in method notation. These methods are described in 4.7.1 through 4.7.15.

4.7.1 Len()

        function int len()

— str.len() returns the length of the string, i.e., the number of characters in the string (excluding any terminating character).
— If str is "", then str.len() returns 0.

4.7.2 Putc()

        task putc(**int** i, **byte** c)

— str.putc(i, c) replaces the *i*th character in *str* with the given integral value.

—   `putc` does not change the size of `str`: If i < 0 or i >= `str.len()`, then *str* is unchanged.

—   if the second argument to `putc` is zero, the string is unaffected.

The putc method assignment `str.putc(j, x)` is semantically equivalent to `str[j] = x`.

4.7.3 Getc()

```
function byte getc(int i)
```

—   `str.getc(i)` returns the ASCII code of the `i`th character in `str`.

—   If i < 0 or i >= `str.len()`, then `str.getc(i)` returns 0.

The getc method assignment `x = str.getc(j)` is semantically equivalent to `x = str[j]`.

4.7.4 Toupper()

```
function string toupper()
```

—   `str.toupper()` returns a string with characters in `str` converted to uppercase.

—   `str` is unchanged.

4.7.5 Tolower()

```
function string tolower()
```

—   `str.tolower()` returns a string with characters in `str` converted to lowercase.

—   `str` is unchanged.

4.7.6 Compare()

```
function int compare(string s)
```

—   `str.compare(s)` compares `str` and `s`, as in the ANSI C `strcmp` function with regard to lexical ordering and return value.

See the relational string operators in 4.7, Table 4-2.

4.7.7 Icompare()

```
function int icompare(string s)
```

—   `str.icompare(s)` compares `str` and `s`, like the ANSI C `strcmp` function with regard to lexical ordering and return value, but the comparison is case insensitive.

4.7.8 Substr()

```
function string substr(int i, int j)
```

—   `str.substr(i, j)` returns a new string that is a substring formed by characters in position `i` through `j` of `str`.

—   If i < 0, j < i, or j >= `str.len()`, `substr()` returns " " (the empty string).

4.7.9 Atoi(), atohex(), atooct(), atobin()

```
function integer atoi()
function integer atohex()
function integer atooct()
function integer atobin()
```

— `str.atoi()` returns the integer corresponding to the ASCII decimal representation in `str`. For
   example:

```
str = "123";
int i = str.atoi();  // assigns 123 to i.
```

The conversion scans all leading digits and underscore characters (_) and stops as soon as it
encounters any other character or the end of the string. It returns zero if no digits were encountered.
It does not parse the full syntax for integer literals (sign, size, tick, base).

— `atohex` interprets the string as hexadecimal.

— `atooct` interprets the string as octal.

— `atobin` interprets the string as binary.

4.7.10 Atoreal()

```
function real atoreal()
```

— `str.atoreal()` returns the real number corresponding to the ASCII decimal representation in `str`.

The conversion parses Verilog syntax for real constants. The scan stops as soon as it encounters any
character that does not conform to this syntax or the end of the string. It returns zero if no digits were
encountered.

4.7.11 Itoa()

```
task itoa(integer i)
```

— `str.itoa(i)` stores the ASCII decimal representation of `i` into `str` (inverse of `atoi`).

4.7.12 Hextoa()

```
task hextoa(integer i)
```

— `str.hextoa(i)` stores the ASCII hexadecimal representation of `i` into `str` (inverse of `atohex`).

4.7.13 Octtoa()

```
task octtoa(integer i)
```

— `str.octtoa(i)` stores the ASCII octal representation of `i` into `str` (inverse of `atooct`).

4.7.14 Bintoa()

```
task bintoa(integer i)
```

— `str.bintoa(i)` stores the ASCII binary representation of `i` into `str` (inverse of `atobin`).

4.7.15 Realtoa()

```
        task realtoa(real r)
```

—   `str.realtoa(r)` stores the ASCII real representation of `r` into `str` (inverse of `atoreal`).

4.8 Event data type

The **event** data type is an enhancement over Verilog named events. SystemVerilog events provide a handle to a synchronization object. As in Verilog, event variables can be explicitly triggered and waited for. Furthermore, SystemVerilog events have a persistent triggered state that lasts for the duration of the entire time step. In addition, an event variable can be assigned another event variable or the special value **null**. When assigned another event variable, both event variables refer to the same synchronization object. When assigned **null**, the association between the synchronization object and the event variable is broken. Events can be passed as arguments to tasks.

The syntax to declare an **event** is as follows:

```
    event variable_name [= initial_value];
```

Where `variable_name` is a valid identifier and the optional **initial_value** can be another event variable or the special value **null**.

If an initial value is not specified, then the variable is initialized to a new synchronization object.

Examples:

```
    event done;              // declare a new event called done
    event done_too = done;   // declare done_too as alias to done
    event empty = null;      // event variable with no synchronization object
```

Event operations and semantics are discussed in detail in 14.5.

4.9 User-defined types

---

type_declaration ::=   // from A.2.1.3
        typedef data_type type_identifier { variable_dimension } **;**
        | **typedef** interface_instance_identifier **.** type_identifier type_identifier **;**
        | **typedef** [**enum** | **struct** | **union** | **class**] type_identifier **;**

Syntax 4-2—User-defined types (excerpt from Annex A)

The user can define a new type using **typedef**, as in C.

```
    typedef int intP;
```

This can then be instantiated as follows:

```
    intP a, b;
```

A type can be used before it is defined, provided it is first identified as a type by an empty **typedef**:

```
typedef foo;

foo f = 1;

typedef int foo;
```

An empty **typedef** shall not be allowed with enumeration values. Enumeration values must be defined before they are used.

User-defined type identifiers have the same scoping rules as data identifiers, except that hierarchical reference to type identifiers shall not be allowed. References to type identifiers defined within an interface through ports are allowed provided they are locally redefined before being used.

```
interface intf_i;

    typedef int data_t;

endinterface


module sub(intf_i p)

    typedef p.data_t my_data_t;

    my_data_t data;

        // type of 'data' will be int when connected to interface above

endmodule
```

User-defined type names must be used for complex data types in casting (see 4.14, below), which only allows simple type names, and as type parameter values when unpacked array types are used.

Sometimes a user-defined type needs to be declared before the contents of the type have been defined. This is of use with user-defined types derived from **enum**, **struct**, **union**, and **class**. For an example, see 7.24. Support for this is provided by the following forms for **typedef**:

```
typedef enum type_identifier;

typedef struct type_identifier;

typedef union type_identifier;

typedef class type_identifier;

typedef type_identifier;
```

While an empty user-defined type declaration is useful for coupled definitions of classes as shown in 7.24, it cannot be used for coupled definitions of structures because structures are statically declared and there is no support for pointers to structures.

The last form shows that the type of the user-defined type does not have to be defined in the forward declaration.

The actual type definition of a forward **typedef** declaration shall be resolved within the same local scope or **generate** block. Importing a **typedef** from a package into a local scope can also resolve a type definition.

4.10 Enumerations

```
data_type ::=                                                                    // from A.2.2.1
      ...
      | enum [ enum_base_type ] { enum_name_declaration { , enum_name_declaration } }
enum_base_type ::=
      integer_atom_type [ signing ]
    | integer_vector_type [ signing ] [ packed_dimension ]
    | type_identifier [ packed_dimension ]24
enum_name_declaration ::=
      enum_identifier [ [ integral_number [ : integral_number ] ] ] [ = constant_expression ]
```

Syntax 4-3—Enumerated types (excerpt from Annex A)

An enumerated type declares a set of integral named constants. Enumerated data types provide the capability to abstractly declare strongly typed variables without either a data type or data value(s) and later add the required data type and value(s) for designs that require more definition. Enumerated data types also can be easily referenced or displayed using the enumerated names as opposed to the enumerated values.

In the absence of a data type declaration, the default data type shall be **int**. Any other data type used with enumerated types shall require an explicit data type declaration.

An enumerated type defines a set of named values. In the following example, light1 and light2 are defined to be variables of the anonymous (unnamed) enumerated **int** type that includes the three members: red, yellow, and green.

```
    enum {red, yellow, green} light1, light2; // anonymous int type
```

An enumerated name with x or z assignments assigned to an **enum** with no explicit data type or an explicit 2-state declaration shall be a syntax error.

```
    // Syntax error: IDLE=2'b00, XX=2'bx <ERROR>, S1=2'b01, S2=2'b10
    enum {IDLE, XX='x, S1=2'b01, S2=2'b10} state, next;
```

An **enum** declaration of a 4-state type, such as integer, that includes one or more names with x or z assignments shall be permitted.

```
    // Correct: IDLE=0, XX='x, S1=1, S2=2
    enum integer {IDLE, XX='x, S1='b01, S2='b10} state, next;
```

An unassigned enumerated name that follows an **enum** name with x or z assignments shall be a syntax error.

```
    // Syntax error: IDLE=2'b00, XX=2'bx, S1=??, S2=??
    enum integer {IDLE, XX='x, S1, S2} state, next;
```

The values can be cast to integer types and increment from an initial value of 0. This can be overridden.

```
    enum {bronze=3, silver, gold} medal; // silver=4, gold=5
```

The values can be set for some of the names and not set for other names. The optional value of an **enum** named constant is an elaboration time constant expression (see 6.3) and can include references to parameters, local parameters, genvars, other **enum** named constants, and constant functions of these. Hierarchical names and **const** variables are not allowed. A name without a value is automatically assigned an increment

of the value of the previous name. It shall be an error if incrementing the previous value causes an overflow within the significant digits of the enumerated type.

```
// c is automatically assigned the increment-value of 8
enum {a=3, b=7, c} alphabet;
```

If an automatically incremented value is assigned elsewhere in the same enumeration, this shall be a syntax error.

```
// Syntax error: c and d are both assigned 8
enum {a=0, b=7, c, d=8} alphabet;
```

If the first name is not assigned a value, it is given the initial value of 0.

```
// a=0, b=7, c=8
enum {a, b=7, c} alphabet;
```

Any enumeration encoding value that is outside the representable range of the **enum** shall be an error. If any of the **enum** members are defined with a different sized constant, this shall be a syntax error.

```
// Correct declaration - bronze and gold are unsized
enum bit [3:0] {bronze='h3, silver, gold='h5} medal4;

// Correct declaration - bronze and gold sizes are redundant
enum bit [3:0] {bronze=4'h3, silver, gold=4'h5} medal4;

// Error in the bronze and gold member declarations
enum bit [3:0] {bronze=5'h13, silver, gold=3'h5} medal4;

// Error in c declaration, requires at least 2 bits
enum bit [0:0] {a,b,c} alphabet;
```

Type checking of enumerated types used in assignments, as arguments, and with operators is covered in 4.10.3. Like C, there is no overloading of literals; therefore, medal and medal4 cannot be defined in the same scope because they contain the same names.

4.10.1 Defining new data types as enumerated types

A type name can be given so that the same type can be used in many places.

```
typedef enum {NO, YES} boolean;
boolean myvar; // named type
```

4.10.2 Enumerated type ranges

A range of enumeration elements can be specified automatically, via the syntax shown in Table 4-3.

Table 4-3—Enumeration element ranges

name	Associates the next consecutive number with name.
name = C	Associates the constant C to name.
name [N]	Generates N named constants in the sequence: name0, name1,..., nameN-1. N shall be a positive integral number.

Table 4-3—Enumeration element ranges *(continued)*

name[N] = C	Optionally, a constant can be assigned to the generated named constants to associate that constant to the first generated named constant; subsequent generated named constants are associated consecutive values. *N* shall be a positive integral number.
`name [N:M]`	Creates a sequence of named constants starting with nameN and incrementing or decrementing until reaching named constant nameM. *N* and *M* shall be nonnegative integral numbers.
name[N:M] = C	Optionally, a constant can be assigned to the generated named constants to associate that constant to the first generated named constants; subsequent generated named constants are associated consecutive values. *N* and *M* shall be nonnegative integral numbers.

For example:

```
typedef enum { add=10, sub[5], jmp[6:8] } E1;
```

This example defines the enumerated type E1, which assigns the number 10 to the enumerated named constant `add`. It also creates the enumerated named constants `sub0`, `sub1`, `sub2`, `sub3`, and `sub4` and assigns them the values 11...15, respectively. Finally, the example creates the enumerated named constants `jmp6`, `jmp7`, and `jmp8` and assigns them the values 16 through 18, respectively.

```
enum { register[2] = 1, register[2:4] = 10 } vr;
```

The example above declares enumerated variable `vr`, which creates the enumerated named constants `register0` and `register1`, which are assigned the values 1 and 2, respectively. Next, it creates the enumerated named constants `register2`, `register3`, and `register4` and assigns them the values 10, 11, and 12.

4.10.3 Type checking

SystemVerilog enumerated types are strongly typed; thus, a variable of type **enum** cannot be directly assigned a value that lies outside the enumeration set unless an explicit cast is used or unless the **enum** variable is a member of a union. This is a powerful type-checking aid that prevents users from accidentally assigning nonexistent values to variables of an enumerated type. The enumeration values can still be used as constants in expressions, and the results can be assigned to any variable of a compatible integral type.

Both the enumeration names and their integer values must be unique. The values can be set to any integral constant value or auto-incremented from an initial value of 0. It is an error to set two values to the same name or to set a value to the same auto-incremented value.

Enumerated variables are type-checked in assignments, arguments, and relational operators. Enumerated variables are auto-cast into integral values, but assignment of arbitrary expressions to an enumerated variable requires an explicit cast.

For example:

```
typedef enum { red, green, blue, yellow, white, black } Colors;
```

This operation assigns a unique number to each of the color identifiers and creates the new data type `Colors`. This type can then be used to create variables of that type.

```
    Colors c;
    c = green;
    c = 1;              // Invalid assignment
    if ( 1 == c )       // OK. c is auto-cast to integer
```

In the example above, the value `green` is assigned to the variable `c` of type `Colors`. The second assignment is invalid because of the strict typing rules enforced by enumerated types.

Casting can be used to perform an assignment of a different data type, or an out-of-range value, to an enumerated type. Casting is discussed in 4.10.4, 4.14, and 4.15.

4.10.4 Enumerated types in numerical expressions

Elements of enumerated type variables can be used in numerical expressions. The value used in the expression is the numerical value associated with the enumerated value. For example:

```
    typedef enum { red, green, blue, yellow, white, black } Colors;

    Colors col;
    integer a, b;

    a = blue * 3;
    col = yellow;
    b = col + green;
```

From the previous declaration, `blue` has the numerical value 2. This example assigns `a` the value of 6 (2*3), and it assigns `b` a value of 4 (3+1).

An **enum** variable or identifier used as part of an expression is automatically cast to the base type of the **enum** declaration (either explicitly or using **int** as the default). A cast shall be required for an expression that is assigned to an **enum** variable where the type of the expression is not equivalent to the enumeration type of the variable.

Casting to an **enum** type shall cause a conversion of the expression to its base type without checking the validity of the value (unless a dynamic cast is used as described in 4.15).

```
    typedef enum {Red, Green, Blue} Colors;
    typedef enum {Mo,Tu,We,Th,Fr,Sa,Su} Week;
    Colors C;
    Week W;
    int I;

    C = Colors'(C+1);           // C is converted to an integer, then added to
                                // one, then converted back to a Colors type

    C = C + 1; C++; C+=2; C = I;  // Illegal because they would all be
                                // assignments of expressions without a cast

    C = Colors'(Su);            // Legal; puts an out of range value into C

    I = C + W;                  // Legal; C and W are automatically cast to int
```

SystemVerilog includes a set of specialized methods to enable iterating over the values of enumerated types.

4.10.4.1 First()

The prototype for the `first()` method is as follows:

```
function enum first();
```

The `first()` method returns the value of the first member of the enumeration.

4.10.4.2 Last()

The prototype for the `last()` method is as follows:

```
function enum last();
```

The `last()` method returns the value of the last member of the enumeration.

4.10.4.3 Next()

The prototype for the `next()` method is as follows:

```
function enum next( int unsigned N = 1 );
```

The `next()` method returns the Nth next enumeration value (default is the next one) starting from the current value of the given variable. A wrap to the start of the enumeration occurs when the end of the enumeration is reached. If the given value is not a member of the enumeration, the `next()` method returns the default initial value for the enumeration.

4.10.4.4 Prev()

The prototype for the `prev()` method is as follows:

```
function enum prev( int unsigned N = 1 );
```

The `prev()` method returns the Nth previous enumeration value (default is the previous one) starting from the current value of the given variable. A wrap to the end of the enumeration occurs when the start of the enumeration is reached. If the given value is not a member of the enumeration, the `prev()` method returns the default initial value for the enumeration.

4.10.4.5 Num()

The prototype for the `num()` method is as follows:

```
function int num();
```

The `num()` method returns the number of elements in the given enumeration.

4.10.4.6 Name()

The prototype for the name() method is as follows:

```
function string name();
```

The `name()` method returns the string representation of the given enumeration value. If the given value is not a member of the enumeration, the `name()` method returns the empty string.

4.10.4.7 Using enumerated type methods

The following code fragment shows how to display the name and value of all the members of an enumeration:

```
typedef enum { red, green, blue, yellow } Colors;
Colors c = c.first;
forever begin
   $display( "%s : %d\n", c.name, c );
   if( c == c.last ) break;
   c = c.next;
end
```

4.11 Structures and unions

data_type ::=   // from A.2.2.1
        ...
    | struct_union [**packed** [signing]] **{** struct_union_member { struct_union_member } **}**
        { packed_dimension }[13]
struct_union_member[26] ::=
        { attribute_instance } [random_qualifier] data_type_or_void list_of_variable_decl_assignments **;**
data_type_or_void ::= data_type | **void**
struct_union ::= **struct** | **union** [**tagged**]

Syntax 4-4—Structures and unions (excerpt from Annex A)

Structure and union declarations follow the C syntax, but without the optional structure tags before the '{'.

```
struct { bit [7:0] opcode; bit [23:0] addr; }IR;   // anonymous structure
                                                   // defines variable IR
IR.opcode = 1;  // set field in IR.
```

Some additional examples of declaring structure and unions are as follows:

```
typedef struct {
            bit [7:0] opcode;
            bit [23:0] addr;
} instruction; // named structure type
instruction IR; // define variable

typedef union { int i; shortreal f; } num; // named union type
num n;
n.f = 0.0; // set n in floating point format

typedef struct {
            bit isfloat;
            union { int i; shortreal f; } n; // anonymous type
} tagged_st; // named structure

tagged_st a[9:0]; // array of structures
```

A structure can be assigned as a whole and passed to or from a function or task as a whole.

Members of a structure data type may be assigned individual default member values by using an initial assignment with the declaration of each member. The assigned expression must be a constant expression.

An example of initializing member of a structure type is as follows:

```
typedef struct {
   int addr = 1 + constant;
   int crc;
   byte data [4] = '{4'{1}};
} packet1;
```

The structure can then be instantiated.

```
packet1 p1;    // initialization defined by the typedef.
               // p1.crc will use the default value for an int
```

If an explicit initial value expression is used with the declaration of a variable, the initial assignment expression within the structure data type shall be ignored. Subclause 3.8 discusses assigning initial values to a structure. For example:

```
packet1 pi = '{1,2,'{2,3,4,5}}; //suppresses the typedef initialization
```

Members of unpacked structures containing a union as well as members of packed structures shall not be assigned individual default member values.

The initial assignment expression within a data type shall be ignored when using a data type to declare a net (see 6.5).

A packed structure is a mechanism for subdividing a vector into subfields that can be conveniently accessed as members. Consequently, a packed structure consists of bit fields, which are packed together in memory without gaps. An unpacked structure has an implementation-dependent packing, normally matching the C compiler. A packed structure differs from an unpacked structure in that, when a packed structure appears as a primary, it is treated as a single vector.

One special provision exists in order to simplify the use of unpacked unions: if an unpacked union contains several unpacked structures that share a common initial sequence (see below) and if the unpacked union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the complete type of the union is visible. Two structures share a common initial sequence if corresponding members have equivalent types for a sequence of one or more initial members.

Like a packed array, a packed structure can be used as a whole with arithmetic and logical operators. The first member specified is the most significant and subsequent members follow in decreasing significance. The structures are declared using the **packed** keyword, which can be followed by the **signed** or **unsigned** keyword, according to the desired arithmetic behavior. The default is unsigned:

```
struct packed signed {
   int a;
   shortint b;
   byte c;
   bit [7:0] d;
} pack1; // signed, 2-state

struct packed unsigned {
   time a;
   integer b;
```

```
        logic [31:0] c;
    } pack2; // unsigned, 4-state
```

If all data types within a packed structure are 2-state, the structure as a whole is treated as a 2-state vector.

If any data type within a packed structure is 4-state, the structure as a whole is treated as a 4-state vector. If there are also 2-state members in the structure, there is an implicit conversion from 4-state to 2-state when reading those members and from 2-state to 4-state when writing them.

One or more bits of a packed structure can be selected as if it were a packed array, assuming an [n-1:0] numbering:

```
    pack1 [15:8] // c
```

Noninteger data types, such as **real** and **shortreal**, are not allowed in packed structures or unions. Neither are unpacked arrays.

A packed structure can be used with a **typedef**.

```
    typedef struct packed { // default unsigned
        bit [3:0] GFC;
        bit [7:0] VPI;
        bit [11:0] VCI;
        bit CLP;
        bit [3:0] PT ;
        bit [7:0] HEC;
        bit [47:0] [7:0] Payload;
        bit [2:0] filler;
    } s_atmcell;
```

A packed union shall contain members that must be packed structures, or packed arrays or integer data types all of the same size (in contrast to an unpacked union, where the members can be different sizes). This ensures that a union member that was written as another member can be read back. A packed union can also be used as a whole with arithmetic and logical operators, and its behavior is determined by the **signed** or **unsigned** keyword, the latter being the default. One or more bits of a packed union can be selected as if it were a packed array, assuming an [n–1:0] numbering.

If a packed union contains a 2-state member and a 4-state member, the entire union is 4-state. There is an implicit conversion from 4-state to 2-state when reading and from 2-state to 4-state when writing the 2-state member.

For example, a union can be accessible with different access widths:

```
    typedef union packed { // default unsigned
        s_atmcell acell;
        bit [423:0] bit_slice;
        bit [52:0][7:0] byte_slice;
    } u_atmcell;

    u_atmcell u1;
    byte b; bit [3:0] nib;
    b = u1.bit_slice[415:408]; // same as b = u1.byte_slice[51];
    nib = u1.bit_slice [423:420]; // same as nib = u1.acell.GFC;
```

With packed unions, writing one member and reading another is independent of the byte ordering of the machine, unlike an unpacked union of unpacked structures, which are C-compatible and have members in ascending address order.

The signing of unpacked structures is not allowed. The following declaration would be considered illegal:

```
typedef struct signed {
   int f1 ;
   logic f2 ;
} sIllegalSignedUnpackedStructType;    // illegal declaration
```

If no initial value is specified in the declaration of a variable of an unpacked union type, then the variable shall be initialized to the default initial value for variables of the type of the first member in declaration order of the union type.

The qualifier **tagged** in a union declares it as a tagged union, which is a type-checked union. An ordinary (untagged) union can be updated using a value of one member type and read as a value of another member type, which is a potential type loophole. A tagged union stores both the member value and a tag, i.e., additional bits representing the current member name. The tag and value can only be updated together consistently using a statically type-checked tagged union expression (see 8.14). The member value can only be read with a type that is consistent with the current tag value (i.e., member name). Thus, it is impossible to store a value of one type and (mis)interpret the bits as another type.

Dynamic types and chandles shall not be used in untagged unions, but may be used in tagged unions.

In addition to type safety, the use of member names as tags also makes code simpler and smaller than code that has to track unions with explicit tags. Tagged unions can also be used with pattern matching (see 10.4), which improves readability even further.

In tagged unions, members can be declared with type **void**, when all the information is in the tag itself, as in the following example of an integer together with a valid bit:

```
typedef union tagged {
   void Invalid;
   int Valid;
} VInt;
```

A value of VInt type is either Invalid (and contains nothing) or Valid (and contains an **int**). Subclause 8.14 describes how to construct values of this type and also describes how it is impossible to read an integer out of a VInt value that currently has the Invalid tag.

For example:

```
typedef union tagged {
   struct {
      bit [4:0] reg1, reg2, regd;
      Add;
   union tagged {
      bit [9:0] JmpU;
      struct {
         bit [1:0] cc;
         bit [9:0] addr;
         JmpC;
      Jmp;
   Instr;
```

A value of `Instr` type is either an `Add` instruction, in which case it contains three 5-bit register fields, or it is a `Jmp` instruction. In the latter case, it is either an unconditional jump, in which case it contains a 10-bit destination address, or it is a conditional jump, in which case it contains a 2-bit condition-code register field and a 10-bit destination address. Subclause 8.14 describes how to construct values of `Instr` type and describes how, in order to read the `cc` field, for example, the instruction must have opcode `Jmp` and sub-opcode `JmpC`.

When the **packed** qualifier is used on a tagged union, all the members must have packed types, but they do not have to be of the same size. The (standard) representation for a packed tagged union is the following:

— The size is always equal to the number of bits needed to represent the tag plus the maximum of the sizes of the members.

— The size of the tag is the minimum number of bits needed to code for all the member names (e.g., 5 to 8 members would need 3 tag bits).

— The tag bits are always left-justified (i.e., towards the MSBs).

— For each member, the member bits are always right-justified [i.e., towards the least significant bits (LSBs)].

— The bits between the tag bits and the member bits are undefined. In the extreme case of a void member, only the tag is significant and all the remaining bits are undefined.

The representation scheme is applied recursively to any nested tagged unions.

For example: If the `VInt` type definition had the **packed** qualifier, `Invalid` and `Valid` values will have the layouts shown in Figure 4-1, respectively.

tag is 0 for Invalid, 1 for Valid
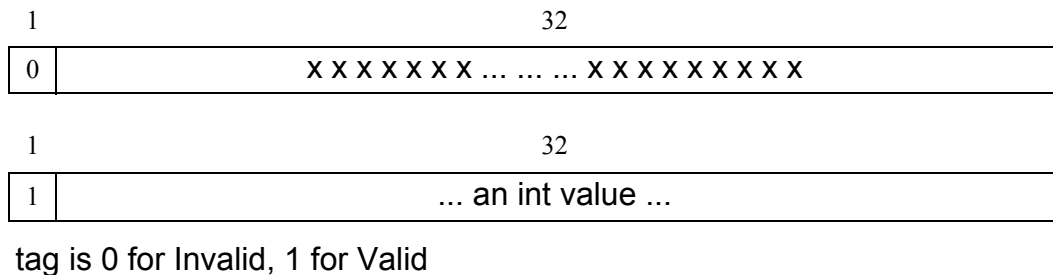
Figure 4-1—VInt type with packed qualifier

For example: If the `Instr` type had the **packed** qualifier, its values will have the layouts shown in Figure 4-2.

1	5	5	5
0	reg1	reg2	regd

Add Instructions

1	2	1	2	10
1	xx	0	xx	

Jmp/JmpU Instructions

1	2	1	2	10
1	xx	1	cc	addr

Jmp/JmpC Instructions
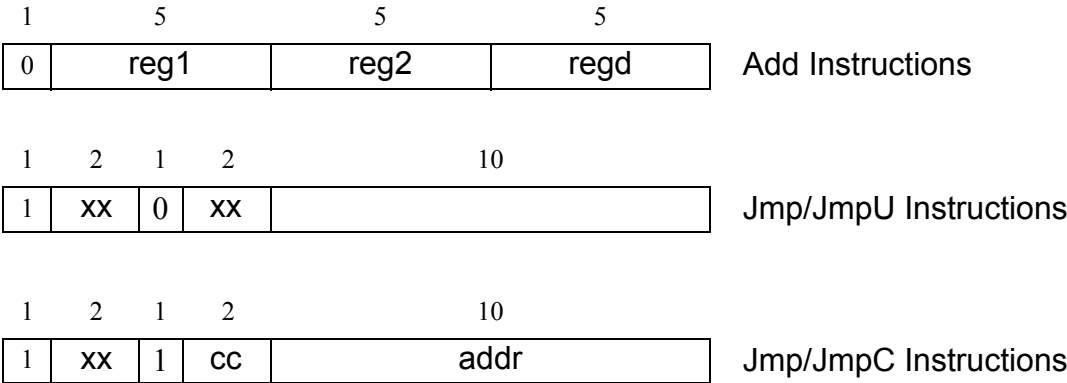
Outer tag is 0 for Add, 1 for Jmp
Inner tag is 0 for JmpU, 1 for JmpC

Figure 4-2—Instr type with packed qualifier

4.12 Class

A class is a collection of data and a set of subroutines that operate on that data. The data in a class are referred to as c*lass properties*, and the subroutines of the class are called *methods*. The class properties and methods, taken together, define the contents and capabilities of a class instance or object.

```
class_declaration ::=                                               // from A.1.2
       [ virtual ] class [ lifetime ] class_identifier [ parameter_port_list ]
           [ extends class_type [ ( list_of_arguments ) ] ];
           { class_item }
       endclass [ : class_identifier]
```

Syntax 4-5—Classes (excerpt from Annex A)

The object-oriented class extension allows objects to be created and destroyed dynamically. Class instances, or objects, can be passed around via object handles, which add a safe-pointer capability to the language. An object can be declared as an argument with direction **input**, **output**, **inout**, or **ref**. In each case, the argument copied is the object handle, not the contents of the object.

A class is declared using the **class**...**endclass** keywords. For example:

```
class Packet;
   int address;        // Properties are address, data, and crc
   bit [63:0] data;
   shortint crc;
   Packet next;        // Handle to another Packet

   function new();     // Methods are send and new
   function bit send();
endclass : Packet
```

Any data type can be declared as a class member. Classes are discussed in more detail in Clause 7.

4.13 Singular and aggregate types

Data types are categorized as either *singular* or *aggregate*. A singular type shall be any data type except an unpacked structure, unpacked union, or unpacked array (see Clause 5 on arrays). An aggregate type shall be any unpacked structure, unpacked union, or unpacked array data type. A singular variable or expression represents a single value, symbols, or handle. Aggregate expressions and variables represent a set or collection of singular values. Integral types are always singular even though they can be sliced into multiple singular values.

These categories are defined so that operators and functions can simply refer to these data types as a collective group. For example, some functions recursively descend into an aggregate variable until reaching a singular value and then perform an operation on each singular value.

Although a class, as described in Clause 7, is a type, there are no variables or expressions of class type directly, only class object handles that are singular. Therefore, classes need not be categorized in this manner (see Clause 7 on classes).

4.14 Casting

```
constant_cast ::=                                                      // from A.8.4
        casting_type ' ( constant_expression )
cast ::=
        casting_type ' ( expression )
casting_type ::= simple_type | constant_primary | signing              // from A.2.2.1
simple_type ::= integer_type | non_integer_type | ps_type_identifier | ps_parameter_identifier
```

Syntax 4-6—Casting (excerpt from Annex A)

A data type can be changed by using a cast (**'**) operation. In a static cast, the expression to be cast shall be enclosed in parentheses that are prefixed with the casting type and an apostrophe. If the expression is assignment compatible with the casting type, then the cast shall return the value that a variable of the casting type would hold after being assigned the expression. If the expression is not assignment compatible with the casting type, then if the casting type is an enumerated type, the behavior shall be as described as in 4.15, and if the casting type is a bit-stream type, the behavior shall be as described in 4.16.

```
int'(2.0 * 3.0)
shortint'{{8'hFA,8'hCE}}
```

A positive decimal number as a data type means a number of bits to change the size.

```
17'(x - 2)
```

The signedness can also be changed.

```
signed'(x)
```

The expression inside the cast must be an integral value when changing the size or signing. When changing the size, the signing shall pass through unchanged and the result type shall be a one-dimensional packed array with a right bound of zero. When changing the signing, the type of the expression to be cast shall pass through unchanged, except for the signing.

When casting to a predefined type, the prefix of the cast must be the predefined type keyword. When casting to a user-defined type, the prefix of the cast must be the user-defined type identifier.

When a **shortreal** is converted to an **int** or to 32 bits, its value is rounded, as in Verilog. Therefore, the conversion can lose information. To convert a **shortreal** to its underlying bit representation without a loss of information, use $shortrealtobits as defined in 22.5. To convert from the bit representation of a shortreal value into a **shortreal**, use $bitstoshortreal as defined in 22.5.

Structures can be converted to bits preserving the bit pattern. In other words, they can be converted back to the same value without any loss of information. When unpacked data are converted to the packed representation, the order of the data in the packed representation is such that the first field in the structure occupies the MSBs. The effect is the same as a concatenation of the data items (**struct** fields or array elements) in order. The type of the elements in an unpacked structure or array must be valid for a packed representation in order to be cast to any other type, whether packed or unpacked.

An explicit cast between packed types is not required because they are implicitly cast as integral values, but a cast can be used by tools to perform stronger type checking.

The following example demonstrates how the $bits attribute is used to obtain the size of a structure in bits (the $bits system function is discussed in 22.3), which facilitates conversion of the structure into a packed array:

```
typedef struct {
           bit isfloat;
           union { int i; shortreal f; } n; // anonymous type
} tagged_st;                               // named structure

typedef bit [$bits(tagged_st) - 1 : 0] tagbits;  // tagged_st defined above

tagged_st a [7:0];                         // unpacked array of structures

tagbits t = tagbits'(a[3]);          // convert structure to array of bits
a[4] = tagged_st'(t);                // convert array of bits back to
structure
```

Note that the **bit** data type loses X values. If these are to be preserved, the **logic** type should be used instead.

The size of a union in bits is the size of its largest member. The size of a **logic** in bits is 1.

For compatibility, the Verilog functions $itor, $rtoi, $bitstoreal, $realtobits, $signed, and $unsigned can also be used.

4.15 $cast dynamic casting

SystemVerilog provides the $cast system task to assign values to variables that might not ordinarily be valid because of differing data type. $cast can be called as either a task or a function.

The syntax for $cast is as follows:

```
function int $cast( singular dest_var, singular source_exp );
```

or

```
task $cast( singular dest_var, singular source_exp );
```

The `dest_var` is the variable to which the assignment is made.

The `source_exp` is the expression that is to be assigned to the destination variable.

Use of `$cast` as either a task or a function determines how invalid assignments are handled.

When called as a task, `$cast` attempts to assign the source expression to the destination variable. If the assignment is invalid, a run-time error occurs, and the destination variable is left unchanged.

When called as a function, `$cast` attempts to assign the source expression to the destination variable and returns 1 if the cast is legal. If the cast fails, the function does not make the assignment and returns 0. When called as a function, no run-time error occurs, and the destination variable is left unchanged.

It is important to note that `$cast` performs a run-time check. No type checking is done by the compiler, except to check that the destination variable and source expression are singulars.

For example:

```
typedef enum { red, green, blue, yellow, white, black } Colors;
Colors col;
$cast( col, 2 + 3 );
```

This example assigns the expression (5 => black) to the enumerated type. Without `$cast` or the static compile-time cast described below in this subclause, this type of assignment is illegal.

The following example shows how to use the `$cast` to check whether an assignment will succeed:

```
if ( ! $cast( col, 2 + 8 ) )     // 10: invalid cast
    $display( "Error in cast" );
```

Alternatively, the preceding examples can be cast using a static SystemVerilog cast operation. For example:

```
col = Colors'(2 + 3);
```

However, this is a compile-time cast, i.e, a coercion that always succeeds at run time and does not provide for error checking or warn if the expression lies outside the enumeration values.

Allowing both types of casts gives full control to the user. If users know that it is safe to assign certain expressions to an enumerated variable, the faster static compile-time cast can be used. If users need to check if the expression lies within the enumeration values, it is not necessary to write a lengthy switch statement manually. The compiler automatically provides that functionality via the `$cast` function. By allowing both types of casts, users can control the time/safety trade-offs.

NOTE—$cast is similar to the `dynamic_cast` function available in C++. However, `$cast` allows users to check whether the operation will succeed, whereas `dynamic_cast` always raises a C++ exception.

4.16 Bit-stream casting

Type casting can also be applied to unpacked arrays and structs. It is thus possible to convert freely between bit-stream types using explicit casts. Types that can be packed into a stream of bits are called *bit-stream types*. A bit-stream type is a type consisting of the following:

— Any integral, packed, or string type
— Unpacked arrays, structures, or classes of the above types
— Dynamically sized arrays (dynamic, associative, or queues) of any of the above types

This definition is recursive so that, for example, a structure containing a queue of **int** is a bit-stream type.

Assuming A is of bit-stream type source_t and B is of bit-stream type dest_t, it is legal to convert A into B by an explicit cast:

```
B = dest_t'(A);
```

The conversion from A of type source_t to B of type dest_t proceeds in two steps:

a) Conversion from source_t to a generic packed value containing the same number of bits as source_t. If source_t contains any 4-state data, the entire packed value is 4-state; otherwise, it is 2-state.

b) Conversion from the generic packed value to dest_t. If the generic packed value is a 4-state type and parts of dest_t designate 2-state types, then those parts in dest_t are assigned as if cast to a 2-state.

When a dynamic array, queue, or string is converted to the packed representation, the item at index 0 occupies the MSBs. When an associative array is converted to the packed representation, items are packed in index-sorted order with the first indexed element occupying the MSBs.

Both source_t and dest_t can include one or more dynamically sized data in any position (for example, a structure containing a dynamic array followed by a queue of bytes). If the source type, source_t, includes dynamically sized variables, they are all included in the bit stream. If the destination type, dest_t, includes unbounded dynamically sized types, the conversion process is greedy: compute the size of the source_t, subtract the size of the fixed-size data items in the destination, and then adjust the size of the first dynamically sized item in the destination to the remaining size; any remaining dynamically sized items are left empty.

For the purposes of a bit-stream cast, a string is considered a dynamic array of bytes.

Regardless of whether the destination type contains only fixed-size items or dynamically sized items, data are extracted into the destination in left-to-right order. It is thus legal to fill a dynamically sized item with data extracted from the middle of the packed representation.

If both source_t and dest_t are fixed-size types of different sizes and either type is unpacked, then a cast generates a compile-time error. If source_t or dest_t contain dynamically sized types, then a difference in their sizes will generate an error either at compile time or at run time, as soon as it is possible to determine the size mismatch. For example:

```
// Illegal conversion from 24-bit struct to int (32 bits) - compile time
error
struct {bit[7:0] a; shortint b;} a;
int b = int'(a);

// Illegal conversion from 20-bit struct to int (32 bits) - run time error
struct {bit a[$]; shortint b;} a = {{1,2,3,4}, 67};
int b = int'(a);

// Illegal conversion from int (32 bits) to struct dest_t (25 or 33 bits),
// compile time error
typedef struct {byte a[$]; bit b;} dest_t;
int a;
dest_t b = dest_t'(a);
```

Bit-stream casting can be used to convert between different aggregate types, such as two structure types, or a structure and an array or queue type. This conversion can be useful to model packet data transmission over

serial communication streams. For example, the code below uses bit-stream casting to model a control packet transfer over a data stream:

```
typedef struct {
   shortint address;
   reg [3:0] code;
   byte command [2];
} Control;

typedef bit Bits [36:1];

Control p;
Bits stream[$];

p = ...                     // initialize control packet
stream = {stream, Bits'(p)}  // append packet to unpacked queue of bits

Control q;
q = Control'(stream[0]);    // convert stream back to a Control packet
stream = stream[1:$];       // remove packet from stream
```

The following example uses bit-stream casting to model a data packet transfer over a byte stream:

```
typedef struct {
   byte length;
   shortint address;
   byte payload[];
   byte chksum;
} Packet;
```

The above type defines a generic data packet in which the size of the payload field is stored in the length field. Below is a function that randomly initializes the packet and computes the checksum.

```
function Packet genPkt();
   Packet p;

   void'( randomize( p.address, p.length, p.payload )
      with { p.length > 1 && p.payload.size == p.length; } );
   p.chksum = p.payload.xor();
   return p;
endfunction
```

The byte stream is modeled using a queue, and a bit-stream cast is used to send the packet over the stream.

```
typedef byte channel_type[$];
channel_type channel;
channel = {channel, channel_type'(genPkt())};
```

And the code to receive the packet:

```
Packet p;
int    size;

size = channel[0] + 4;
p = Packet'( channel[0 : size - 1] );    // convert stream to Packet
channel = channel[ size, $ ];             // remove packet data from stream
```

4.17 Default attribute type

The default type of an attribute with no value is `bit`, with a value of 1. Otherwise, the attribute takes the type of the expression.

NOTE—With Verilog, users can add named attributes (properties) to Verilog objects, such as modules, instances, and wires. Attributes can also be specified on the extended SystemVerilog constructs and are included as part of the BNF (see Annex A). SystemVerilog also defines a default data type for attributes.

5. Arrays

5.1 Introduction

NOTE—An array is a collection of variables, all of the same type, and accessed using the same name plus one or more indices.

In C, arrays are indexed from 0 by integers or converted to pointers. Although the whole array can be initialized, each element must be read or written separately in procedural statements.

In Verilog, arrays are indexed from left-bound to right-bound. If they are vectors, they can be assigned as a single unit, but not if they are arrays. Verilog allows multiple dimensions.

In Verilog, all data types can be declared as arrays. The **reg** type, the **wire** type, and all other net types can also have a vector width declared. A dimension declared before the object name is referred to as the *vector width dimension*. The dimensions declared after the object name are referred to as the *array dimensions*.

```
reg [7:0] r1 [1:256];   // [7:0] is the vector width, [1:256] is the array
size
```

SystemVerilog uses the term *packed array* to refer to the dimensions declared before the object name (what Verilog refers to as the vector width). The term *unpacked array* is used to refer to the dimensions declared after the object name.

```
bit [7:0] c1;         // packed array
real u [7:0];         // unpacked array
```

SystemVerilog enhances packed arrays by allowing multiple dimensions. SystemVerilog adds the ability to procedurally change the size of one of the dimensions of an unpacked array. Fixed-size unpacked arrays can be multi-dimensional and have fixed storage allocated for all the elements of the array. Each dimension of an unpacked array can be declared as having a fixed or unfixed size. A dynamic array allocates storage for elements at run time along with the option of changing the size of one of its dimensions. An associative array allocates storage for elements individually as they are written. Associative arrays can be indexed using arbitrary data types. A queue type of array grows or shrinks to accommodate the number of elements written to the array at run time.

5.2 Packed and unpacked arrays

A packed array is a mechanism for subdividing a vector into subfields that can be conveniently accessed as array elements. Consequently, a packed array is guaranteed to be represented as a contiguous set of bits. An unpacked array may or may not be so represented. A packed array differs from an unpacked array in that, when a packed array appears as a primary, it is treated as a single vector.

If a packed array is declared as signed, then the array viewed as a single vector shall be signed. The individual elements of the array are unsigned unless they are of a named type declared as signed. A part-select of a packed array shall be unsigned.

Packed arrays allow arbitrary length integer types; therefore, a 48-bit integer can be made up of 48 bits. These integers can then be used for 48-bit arithmetic. The maximum size of a packed array can be limited, but shall be at least 65 536 (2^{16}) bits.

Packed arrays can be made of only the single bit data types (**bit**, **logic**, **reg**) and recursively other packed arrays and packed structures.

Integer types with predefined widths cannot have packed array dimensions declared. These types are **byte**, **shortint**, **int**, **longint**, and **integer**. Although an integer type with a predefined width n is not a packed array, it matches (see 6.9.2), and can be selected from as if it were, a packed array type with a single [n-1:0] dimension.

```
byte c2;    // same as bit signed [7:0] c2;
integer i1; // same as logic signed [31:0] i1;
```

Unpacked arrays can be made of any data type. Each dimension of an unpacked array can be declared as having a fixed or unfixed size. Fixed-size unpacked arrays can be multidimensional and have fixed storage allocated for all the elements of the array. If an unpacked array has one or more dynamic, associative, or queued dimensions, it is considered a variable-size array.

SystemVerilog accepts a single positive number, as an alternative to a range, to specify the size of an unpacked array, like C. In other words, `[size]` becomes the same as `[0:size-1]`. For example:

    int `Array[8][32];` is the same as **int** `Array[0:7][0:31];`

The following operations can be performed on all arrays, packed or unpacked. The examples provided with these rules assume that `A` and `B` are arrays of the same shape and type.

— Reading and writing the array, e.g., `A = B`
— Reading and writing a slice of the array, e.g., `A[i:j] = B[i:j]`
— Reading and writing a variable slice of the array, e.g., `A[x+:c] = B[y+:c]`
— Reading and writing an element of the array, e.g., `A[i] = B[i]`
— Equality operations on the array or slice of the array, e.g., `A==B`, `A[i:j] != B[i:j]`

The following operations can be performed on packed arrays, but not on unpacked arrays. The examples provided with these rules assume that `A` is an array.

— Assignment from an integer, e.g., `A = 8'b11111111;`
— Treatment as an integer in an expression, e.g., `(A + 3)`

If an unpacked array is declared as signed, then this applies to the individual elements of the array because the whole array cannot be viewed as a single vector.

When assigning to an unpacked array, the source and target must be arrays with the same number of unpacked dimensions, and the length of each dimension must be the same. Assignment to an unpacked array is done by assigning each element of the source unpacked array to the corresponding element of the target unpacked array. The leftmost element of the source array corresponds to the leftmost element of the target array. Each element of an unpacked array that is assigned to the corresponding element of another unpacked array can itself be a packed array.

For the purposes of assignment, a packed array is treated as a vector. Any vector expression can be assigned to any packed array. The packed array bounds of the target packed array do not affect the assignment. A packed array cannot be directly assigned to an unpacked array without an explicit cast.

5.3 Multiple dimensions

Like Verilog memories, the dimensions preceding the identifier set the packed size. The dimensions following the identifier set the unpacked size.

```
bit [3:0] [7:0] joe [1:10]; // 10 entries of 4 bytes (packed into 32 bits)
```

can be used as follows:

```
joe[9] = joe[8] + 1; // 4 byte add
joe[7][3:2] = joe[6][1:0]; // 2 byte copy
```

In a multidimensional declaration, the dimensions declared following the type and before the name (`[3:0][7:0]` in the preceding declaration) vary more rapidly than the dimensions following the name (`[1:10]` in the preceding declaration). When referenced, the packed dimensions (`[3:0]`, `[7:0]`) follow the unpacked dimensions (`[1:10]`).

In a list of dimensions, the rightmost one varies most rapidly, as in C. However, a packed dimension varies more rapidly than an unpacked one.

```
bit [1:10] foo1 [1:5];   // 1 to 10 varies most rapidly; compatible with
                              Verilog arrays
bit foo2 [1:5] [1:10];   // 1 to 10 varies most rapidly, compatible with C

bit [1:5] [1:10] foo3;   // 1 to 10 varies most rapidly

bit [1:5] [1:6] foo4 [1:7] [1:8];    // 1 to 6 varies most rapidly, followed by
                                         1 to 5, then 1 to 8 and then 1 to 7
```

Multiple packed dimensions can also be defined in stages with **typedef**.

```
typedef bit [1:5] bsix;
bsix [1:10] foo5; // 1 to 5 varies most rapidly
```

Multiple unpacked dimensions can also be defined in stages with **typedef**.

```
typedef bsix mem_type [0:3];  // array of four 'bsix' elements
mem_type bar [0:7];           // array of eight 'mem_type' elements
```

When the array is used with a smaller number of dimensions, these have to be the slowest varying ones.

```
bit [9:0] foo6;
foo6 = foo1[2]; // a 10-bit quantity.
```

As in Verilog, a comma-separated list of array declarations can be made. All arrays in the list shall have the same data type and the same packed array dimensions.

```
bit [7:0] [31:0] foo7 [1:5] [1:10], foo8 [0:255]; // two arrays declared
```

If an index expression is out of the address bounds or if any bit in the address is X or Z, then the index shall be invalid. The result of reading from an array with an invalid index shall return the default uninitialized value for the array element type. Writing to an array with an invalid index shall perform no operation. Implementations can generate a warning if an invalid index occurs for a read or write operation of an array.

5.4 Indexing and slicing of arrays

An expression can select part of a packed array, or any integer type, which is assumed to be numbered down to 0.

SystemVerilog uses the term *part-select* to refer to a selection of one or more contiguous bits of a single-dimension packed array. This is consistent with the usage of the term *part-select* in Verilog.

```
reg [63:0] data;
reg [7:0] byte2;
byte2 = data[23:16]; // an 8-bit part-select from data
```

SystemVerilog uses the term *slice* to refer to a selection of one or more contiguous elements of an array. Verilog only permits a single element of an array to be selected and does not have a term for this selection.

A single element of a packed or unpacked array can be selected using an indexed name.

```
bit [3:0] [7:0] j;  // j is a packed array
byte k;
```

```
k = j[2]; // select a single 8-bit element from j
```

One or more contiguous elements can be selected using a slice name. A slice name of a packed array is a packed array. A slice name of an unpacked array is an unpacked array.

```
bit busA [7:0] [31:0] ;    // unpacked array of 8 32-bit vectors
int busB [1:0];            // unpacked array of 2 integers
busB = busA[7:6];          // select a slice from busA
```

The size of the part-select or slice must be constant, but the position can be variable. The syntax of Verilog is used.

```
int i = bitvec[j +: k];   // k must be constant.
int a[x:y], b[y:z], e;
a = {b[c -: d], e};       // d must be constant
```

Slices of an array can only apply to one dimension, but other dimensions can have single index values in an expression.

5.5 Array querying functions

SystemVerilog provides new system functions to return information about an array. These are **$left**, **$right**, **$low**, **$high**, **$increment**, **$size**, **$dimensions**, and **$unpacked_dimensions**. These functions are described in 22.6.

5.6 Dynamic arrays

A dynamic array is any dimension of an unpacked array whose size can be set or changed at run time. The space for a dynamic array does not exist until the array is explicitly created at run time.

The syntax to declare a dynamic array is as follows:

```
data_type array_name [];
```

where `data_type` is the data type of the array elements. Dynamic arrays support the equivalent types as fixed-size arrays.

For example:

```
bit [3:0] nibble[];    // Dynamic array of 4-bit vectors
integer mem[];         // Dynamic array of integers
```

The **new**[] operator is used to set or change the size of the array.

The `size()` built-in method returns the current size of the array.

The `delete()` built-in method clears all the elements yielding an empty array (zero size).

5.6.1 New[]

The built-in function **new** allocates the storage and initializes the newly allocated array elements either to their default initial value or to the values provided by the optional argument.

The prototype of the **new** function is as follows:

```
blocking_assignment ::=                                                  // from A.6.2
       ...
     | hierarchical_dynamic_array_variable_identifier = dynamic_array_new
       ...
dynamic_array_new ::=                                                     // from A.2.4
       new [ expression ] [ ( expression ) ]
```

Syntax 5-1—Declaration of dynamic array new (excerpt from Annex A)

[expression **]**:

> The number of elements in the array. Must be a non-negative integral expression.

(expression **)**:

> Optional. An array with which to initialize the new array. If it is not specified, the elements of the newly allocated array are initialized to their default value. This array identifier must be a dynamic array of a data type equivalent to the array on the left-hand side, but it need not have the same size. If the size of this array is less than the size of the new array, the extra elements shall be initialized to their default value. If the size of this array is greater than the size of the new array, the additional elements shall be ignored.

> This argument is useful when growing or shrinking an existing array. In this situation, the value of **(** expression **)** is the same as the left-hand side; therefore, the previous values of the array elements are preserved. For example:

```
integer addr[];    // Declare the dynamic array.
addr = new[100];   // Create a 100-element array.
...
    // Double the array size, preserving previous values.
addr = new[200](addr);
```

The **new** operator follows the SystemVerilog precedence rules. Because both the square brackets `[]` and the parenthesis `()` have the same precedence, the arguments to this operator are evaluated left to right: **[** expression **]** first, and **(** expression **)** second.

5.6.2 Size()

The prototype for the `size()` method is as follows:

```
function int size();
```

The `size()` method returns the current size of a dynamic array or returns zero if the array has not been created.

```
int j = addr.size;
addr = new[ addr.size() * 4 ] (addr);  // quadruple addr array
```

The `size` dynamic array method is equivalent to `$size( addr, 1 )` array query system function (see 22.6).

5.6.3 Delete()

The prototype for the `delete()` method is as follows:

```
function void delete();
```

The `delete()` method empties the array, resulting in a zero-sized array.

```
int ab [] = new[ N ];        // create a temporary array of size N
// use ab
ab.delete;                   // delete the array contents
$display( "%d", ab.size );   // prints 0
```

5.7 Array assignment

Assigning to a fixed-size unpacked array requires that the source and the target both be arrays with the same number of unpacked dimensions, the length of each dimension be the same, and each element be of an equivalent type. The same requirements shall be in effect if either or both of the arrays are slices. Assignment is done by assigning each element of the source array to the corresponding element of the target array. Element correspondence is defined as leftmost to leftmost, rightmost to rightmost, irrespective of index values. For example, if array A is declared as `int` A[7:0] and array B is declared as `int` B[1:8], the assignment A = B; will assign element B[1] to element A[7], and so on. Assigning fixed-size unpacked arrays of nonequivalent type to one another shall result in a compiler error. See 6.9.2.

```
int A[10:1];        // fixed-size array of 10 elements
int B[0:9];         // fixed-size array of 10 elements
int C[24:1];        // fixed-size array of 24 elements

A = B;              // ok. Compatible type and same size
A = C;              // type check error: different sizes
```

An array of wires can be assigned to an array of variables, and vice versa, if they have the same number of unpacked dimensions, the same number of elements for each of those dimensions, and an equivalent type of elements. Assignment is done by assigning each element of the source array to the corresponding element of the target array.

```
wire [31:0] W [9:0];
assign W = A;
initial #10 B = W;
```

A dynamic array can be assigned to a fixed-size array of an equivalent type if the size of the dynamic array dimension is the same as the length of the fixed-size array dimension. Unlike assigning with a fixed-size array, this operation requires a run-time check that can result in an error, in which case no operation shall be performed.

```
int A[100:1];          // fixed-size array of 100 elements
int B[] = new[100];    // dynamic array of 100 elements
int C[] = new[8];      // dynamic array of 8 elements

A = B;                 // OK. Compatible type and same size
A = C;                 // type check error: different sizes
```

A dynamic array or a one-dimensional fixed-size array can be assigned to a dynamic array of a compatible type. In this case, the assignment creates a new dynamic array with a size equal to the length of the fixed-size array. For example:

```
int A[100:1];          // fixed-size array of 100 elements
int B[];               // empty dynamic array
int C[] = new[8];      // dynamic array of size 8
```

```
B = A;                      // ok. B has 100 elements
B = C;                      // ok. B has 8 elements
```

The last statement above is equivalent to:

```
B = new[ C.size ] (C);
```

Similarly, the source of an assignment can be a complex expression involving array slices or concatenations. For example:

```
string d[1:5] = '{ "a", "b", "c", "d", "e" };
string p[];
p = { d[1:3], "hello", d[4:5] };
```

The preceding example creates the dynamic array `p` with contents `"a"`, `"b"`, `"c"`, `"hello"`, `"d"`, `"e"`.

5.8 Arrays as arguments

Arrays can be passed as arguments to tasks or functions. The rules that govern array argument passing by value are the same as for array assignment (see 12.4). When an array argument is passed by value, a copy of the array is passed to the called task or function. This is true for all array types: fixed-size, dynamic, or associative.

If one dimension of a formal is unsized (unsized dimensions can occur in dynamic arrays and in formal arguments of import DPI functions), then any size of the corresponding dimension of an actual is accepted.

For example, the declaration

```
task fun(int a[3:1][3:1]);
```

declares task `fun` that takes one argument, a two-dimensional array with each dimension of size 3. A call to `fun` must pass a two-dimensional array and with the same dimension size 3 for all the dimensions. For example, given the above description for `fun`, consider the following actuals:

```
int b[3:1][3:1];      // OK: same type, dimension, and size

int b[1:3][0:2];      // OK: same type, dimension, & size (different ranges)

reg b[3:1][3:1];      // error: incompatible element type

event b[3:1][3:1];    // error: incompatible type

int b[3:1];           // error: incompatible number of dimensions

int b[3:1][4:1];      // error: incompatible size (3 vs. 4)
```

A subroutine that accepts a one-dimensional fixed-size array can also be passed a dynamic array of a compatible type of the same size.

For example, the declaration

```
task bar( string arr[4:1] );
```

declares a task that accepts one argument, an array of 4 strings. This task can accept the following actual arguments:

```
    string b[4:1];        // OK: same type and size
    string b[5:2];        // OK: same type and size (different range)
    string b[] = new[4];  // OK: same type and size, requires run-time check
```

A subroutine that accepts a dynamic array can be passed a dynamic array of a compatible type or a one-dimensional fixed-size array of a compatible type.

For example, the declaration

```
    task foo( string arr[] );
```

declares a task that accepts one argument, a dynamic array of strings. This task can accept any one-dimensional array of strings or any dynamic array of strings.

An import DPI function that accepts a one-dimensional array can be passed a dynamic array of a compatible type and of any size if formal is unsized and of the same size if formal is sized. However, a dynamic array cannot be passed as an argument if formal is an unsized output.

5.9 Associative arrays

Dynamic arrays are useful for dealing with contiguous collections of variables whose number changes dynamically. When the size of the collection is unknown or the data space is sparse, an associative array is a better option. Associative arrays do not have any storage allocated until it is used, and the index expression is not restricted to integral expressions, but can be of any type.

An associative array implements a lookup table of the elements of its declared type. The data type to be used as an index serves as the lookup key and imposes an ordering.

The syntax to declare an associative array is as follows:

```
    data_type array_id [ index_type ];
```

where
data_type	is the data type of the array elements. Can be any type allowed for fixed-size arrays.
array_id	is the name of the array being declared.
index_type	is the data-type to be used as an index or is *****. If ***** is specified, then the array is indexed by any integral expression of arbitrary size. An index type restricts the indexing expressions to a particular type. It shall be illegal for index_type to declare a type.

Examples of associative array declarations are as follows:

```
    integer i_array[*];            // associative array of integer (unspecified
                                   // index)

    bit [20:0] array_b[string];    // associative array of 21-bit vector,
                                   // indexed by string

    event ev_array[myClass];       // associative array of event indexed by class
                                   // myClass
```

Array elements in associative arrays are allocated dynamically; an entry is created the first time it is written. The associative array maintains the entries that have been assigned values and their relative order according to the index data type. Associative array elements are unpacked. In other words, other than for copying or

comparing arrays, an individual element must be selected out of the array before it can be used in most expressions.

5.9.1 Wildcard index type

For example:

```
int array_name [*];
```

Associative arrays that specify a wildcard index type have the following properties:

— The array can be indexed by any integral data type. Because the indices can be of different sizes, the same numerical value can have multiple representations, each of a different size. SystemVerilog resolves this ambiguity by detecting the number of leading zeros and computing a unique length and representation for every value.
— Nonintegral index types are illegal and result in a type check error.
— A 4-state index containing X or Z is invalid.
— Indices are unsigned.
— Indexing expressions are self-determined; signed indices are not sign extended.
— A string literal index is auto-cast to a bit vector of equivalent size.
— The ordering is numerical (smallest to largest).

5.9.2 String index

For example:

```
int array_name [ string ];
```

Associative arrays that specify a string index have the following properties:

— Indices can be strings or string literals of any length. Other types are illegal and shall result in a type check error.
— An empty string "" index is valid.
— The ordering is lexicographical (lesser to greater).

5.9.3 Class index

For example:

```
int array_name [ some_Class ];
```

Associative arrays that specify a class index have the following properties:

— Indices can be objects of that particular type or derived from that type. Any other type is illegal and shall result in a type check error.
— A null index is valid.
— The ordering is deterministic but arbitrary.

5.9.4 Integer (or int) index

For example:

```
int array_name [ integer ];
```

Associative arrays that specify an integer index have the following properties:

— Indices can be any integral expression.
— Indices are signed.
— A 4-state index containing X or Z is invalid.
— Indices smaller than integer are sign extended to 32 bits.
— Indices larger than integer are truncated to 32 bits.
— The ordering is signed numerical.

5.9.5 Signed packed array

For example:

```
typedef bit signed [4:1] Nibble;
int array_name [ Nibble ];
```

Associative arrays that specify a signed packed array index have the following properties:

— Indices can be any integral expression.
— Indices are signed.
— Indices smaller than the size of the index type are sign extended.
— Indices larger than the size of the index type are truncated to the size of the index type.
— The ordering is signed numerical.

5.9.6 Unsigned packed array or packed struct

For example:

```
typedef bit [4:1] Nibble;
int array_name [ Nibble ];
```

Associative arrays that specify an unsigned packed array index have the following properties:

— Indices can be any integral expression.
— Indices are unsigned.
— A 4-state index containing X or Z is invalid.
— Indices smaller than the size of the index type are zero filled.
— Indices larger than the size of the index type are truncated to the size of the index type.
— The ordering is numerical.

If an invalid index (i.e., 4-state expression has Xs) is used during a read operation or an attempt is made to read a nonexistent entry, then a warning is issued; and the default initial value for the array type is returned, as shown in Table 5-1.

If an invalid index is used during a write operation, the write is ignored, and a warning is issued.

Table 5-1—Value read from a nonexistent associative array entry

Type of array	Value read
4-state integral type	'X
2-state integral type	'0
enumeration	base type default initial value
string	""
class	null
event	null

5.9.7 Other user-defined types

For example:

```
typedef struct {real R; int I[*];} Unpkt;
int array_name [ Unpkt ];
```

In general, associative arrays that specify an index of any type have the following properties:

— Declared indices must have the equality operator defined for its type to be legal. This includes all of the dynamically sized types as legal index types

— An index expression that is or contains X or Z in any of its elements is invalid.

— An index expression that is or contains an empty value or null for any of its elements does not make the index invalid.

— If the relational operator is defined for the index type, the ordering is as defined in the preceding clauses. If not, the relative ordering of any two entries in such an associative array can vary, even between successive runs of the same tool. However, the relative ordering must remain the same within the same simulation run while no indices have been added or deleted.

5.10 Associative array methods

In addition to the indexing operators, several built-in methods are provided that allow users to analyze and manipulate associative arrays, as well as iterate over its indices or keys.

5.10.1 Num()

The syntax for the num() method is as follows:

```
function int num();
```

The num() method returns the number of entries in the associative array. If the array is empty, it returns 0.

```
int imem[*];
imem[ 2'b3 ] = 1;
imem[ 16'hffff ] = 2;
imem[ 4b'1000 ] = 3;
$display( "%0d entries\n", imem.num );   // prints "3 entries"
```

5.10.2 Delete()

The syntax for the `delete()` method is as follows:

```
function void delete( [input index] );
```

where `index` is an optional index of the appropriate type for the array in question.

If the `index` is specified, then the `delete()` method removes the entry at the specified index. If the entry to be deleted does not exist, the method issues no warning.

If the `index` is not specified, then the `delete()` method removes all the elements in the array.

```
int map[ string ];
map[ "hello" ] = 1;
map[ "sad" ] = 2;
map[ "world" ] = 3;
map.delete( "sad" );      // remove entry whose index is "sad" from "map"
map.delete;               // remove all entries from the associative array "map"
```

5.10.3 Exists()

The syntax for the `exists()` method is as follows:

```
function int exists( input index );
```

where `index` is an index of the appropriate type for the array in question.

The `exists()` function checks whether an element exists at the specified index within the given array. It returns 1 if the element exists; otherwise, it returns 0.

```
if ( map.exists( "hello" ))
   map[ "hello" ] += 1;
else
   map[ "hello" ] = 0;
```

5.10.4 First()

The syntax for the `first()` method is as follows:

```
function int first( ref index );
```

where `index` is an index of the appropriate type for the array in question.

The `first()` method assigns to the given index variable the value of the first (smallest) index in the associative array. It returns 0 if the array is empty; otherwise, it returns 1.

```
string s;
if ( map.first( s ) )
   $display( "First entry is : map[ %s ] = %0d\n", s, map[s] );
```

5.10.5 Last()

The syntax for the `last()` method is as follows:

```
function int last( ref index );
```

where `index` is an index of the appropriate type for the array in question.

The `last()` method assigns to the given index variable the value of the last (largest) index in the associative array. It returns 0 if the array is empty; otherwise, it returns 1.

```
string s;
if ( map.last( s ) )
    $display( "Last entry is : map[ %s ] = %0d\n", s, map[s] );
```

5.10.6 Next()

The syntax for the `next()` method is as follows:

```
function int next( ref index );
```

where `index` is an index of the appropriate type for the array in question.

The `next()` method finds the entry whose index is greater than the given index. If there is a next entry, the index variable is assigned the index of the next entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0.

```
string s;
if ( map.first( s ) )
    do
        $display( "%s : %d\n", s, map[ s ] );
    while ( map.next( s ) );
```

5.10.7 Prev()

The syntax for the `prev()` method is as follows:

```
function int prev( ref index );
```

where `index` is an index of the appropriate type for the array in question.

The `prev()` function finds the entry whose index is smaller than the given index. If there is a previous entry, the index variable is assigned the index of the previous entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0.

```
string s;
if ( map.last( s ) )
    do
        $display( "%s : %d\n", s, map[ s ] );
    while ( map.prev( s ) );
```

If the argument that was passed to any of the four associative array traversal methods `first`, `last`, `next`, and `prev` is smaller than the size of the corresponding index, then the function returns −1 and shall copy only as much data as can fit into the argument. For example:

```
string   aa[*];
byte     ix;
int      status;
aa[ 1000 ] = "a";
status = aa.first( ix );
    // status is -1
    // ix is 232 (least significant 8 bits of 1000)
```

5.11 Associative array assignment

Associative arrays can be assigned only to another associative array of a compatible type and with the same index type. Other types of arrays cannot be assigned to an associative array, nor can associative arrays be assigned to other types of arrays, whether fixed-size or dynamic.

Assigning an associative array to another associative array causes the target array to be cleared of any existing entries, and then each entry in the source array is copied into the target array.

5.12 Associative array arguments

Associative arrays can be passed as arguments only to associative arrays of a compatible type and with the same index type. Other types of arrays, whether fixed-size or dynamic, cannot be passed to subroutines that accept an associative array as an argument. Likewise, associative arrays cannot be passed to subroutines that accept other types of arrays.

Passing an associative array by value causes a local copy of the associative array to be created.

5.13 Associative array literals

Associative array literals use the `'{index:value}` syntax with an optional default index. Like all other arrays, an associative array can be written one entry at a time, or the whole array contents can be replaced using an array literal.

For example:

```
// an associative array of strings indexed by 2-state integers,
// default is "foo".
string words [int] = '{default: "foo"};

// an associative array of 4-state integers indexed by strings, default is –1.
integer tab [string] = '{"Peter":20, "Paul":22, "Mary":23, default:-1 };
```

If a default value is specified, then reading a nonexistent element shall yield the specified default value. Otherwise, the default initial value as described in Table 5-1 shall be returned.

5.14 Queues

A queue is a variable-size, ordered collection of homogeneous elements. A queue supports constant-time access to all its elements as well as constant-time insertion and removal at the beginning or the end of the queue. Each element in a queue is identified by an ordinal number that represents its position within the queue, with 0 representing the first, and $ representing the last. A queue is analogous to a one-dimensional unpacked array that grows and shrinks automatically. Thus, like arrays, queues can be manipulated using the indexing, concatenation, slicing operator syntax, and equality operators.

Queues are declared using the same syntax as unpacked arrays, but specifying $ as the array size. The maximum size of a queue can be limited by specifying its optional right bound (last index).

```
variable_dimension ::=                                          // from A.2.5
      unsized_dimension
    | unpacked_dimension
    | associative_dimension
    | queue_dimension
queue_dimension ::= [ $ [ : constant_expression ] ]
```

Syntax 5-2—Declaration of queue dimension (excerpt from Annex A)

constant_expression must evaluate to a positive integer value.

For example:

```
byte q1[$];                       // A queue of bytes
string names[$] = { "Bob" };      // A queue of strings with one element
integer Q[$] = { 3, 2, 7 };       // An initialized queue of integers
bit q2[$:255];                    // A queue whose maximum size is 256 bits
```

The empty array literal {} is used to denote an empty queue. If an initial value is not provided in the declaration, the queue variable is initialized to the empty queue.

5.14.1 Queue operators

Queues and dynamic arrays have the same assignment and argument passing semantics. Also, queues support the same operations that can be performed on unpacked arrays and use the same operators and rules except as defined below:

```
int q[$] = { 2, 4, 8 };
int p[$];
int e, pos;

e = q[0];          // read the first (leftmost) item
e = q[$];          // read the last (rightmost) item
q[0] = e;          // write the first item
p = q;             // read and write entire queue (copy)

q = { q, 6 };      // insert '6' at the end (append 6)
q = { e, q };      // insert 'e' at the beginning (prepend e)

q = q[1:$];        // delete the first (leftmost) item
q = q[0:$-1];      // delete the last (rightmost) item
q = q[1:$-1];      // delete the first and last items

q = {};            // clear the queue (delete all items)

q = { q[0:pos-1], e, q[pos,$] };   // insert 'e' at position pos
q = { q[0:pos], e, q[pos+1,$] };   // insert 'e' after position pos
```

Unlike arrays, the empty queue, {}, is a valid queue and the result of some queue operations. The following rules govern queue operators:

— Q[a : b] yields a queue with b - a + 1 elements.
    — If a > b, then Q[a:b] yields the empty queue {}.
    — Q[n : n] yields a queue with one item, the one at position n. Thus, Q[n : n] === { Q[n] }.

— If n lies outside Q's range (n < 0 or n > $), then Q[n:n] yields the empty queue {}.

— If either a or b are 4-state expressions containing X or Z values, it yields the empty queue {}.

— Q[a : b] where a < 0 is the same as Q[0 : b].

— Q[a : b] where b > $ is the same as Q[a : $].

— An invalid index value (i.e., a 4-state expression with Xs or Zs, or a value that lies outside 0...$) shall cause a read operation (e = Q[n]) to return the default initial value for the type of queue item (as described in Table 4-1).

— An invalid index (i.e., a 4-state expression with Xs or Zs, or a value that lies outside 0...$+1) shall cause a write operation to be ignored and a run-time warning to be issued; however, writing to Q[$+1] is legal.

— A queue declared with a right bound [$:N] shall be limited to the indexes 0 through *N* (its maximum size will be *N*+1). An index that lies outside these limits shall be invalid; therefore, a write operation past the end of the queue shall be ignored and issue a warning. The warning can be issued at either compile time or run time, as soon as it is possible to determine that the index lies outside the queue limit.

5.14.2 Queue methods

In addition to the array operators, queues provide several built-in methods. Assume these declarations for the examples that follow:

```
typedef mytype element_t;  // mytype is any legal type for a queue
typedef element_t queue_t[$];
element_t e;
queue_t Q;
int i;
```

5.14.2.1 Size()

The prototype for the size() method is as follows:

```
function int size();
```

The size() method returns the number of items in the queue. If the queue is empty, it returns 0.

```
for ( int j = 0; j < Q.size; j++ ) $display( Q[j] );
```

5.14.2.2 Insert()

The prototype of the insert() method is as follows:

```
function void insert(input int index, input element_t item);
```

The insert() method inserts the given item at the specified index position.

```
Q.insert(i, e) is equivalent to: Q = {Q[0:i-1], e, Q[i,$]}
```

5.14.2.3 Delete()

The prototype of the delete() method is as follows:

```
function void delete(int index);
```

The delete() method deletes the item at the specified index position.

```
Q.delete(i) is equivalent to: Q = {Q[0:i-1], Q[i+1,$]}
```

5.14.2.4 Pop_front()

The prototype of the `pop_front()` method is as follows:

```
function element_t pop_front();
```

The `pop_front()` method removes and returns the first element of the queue.

```
e = Q.pop_front() is equivalent to: e = Q[0]; Q = Q[1,$]
```

5.14.2.5 Pop_back()

The prototype of the `pop_back()` method is as follows:

```
function element_t pop_back();
```

The `pop_back()` method removes and returns the last element of the queue.

```
e = Q.pop_back() is equivalent to: e = Q[$]; Q = Q[0,$-1]
```

5.14.2.6 Push_front()

The prototype of the `push_front()` method is as follows:

```
function void push_front(input element_t item);
```

The `push_front()` method inserts the given element at the front of the queue.

```
Q.push_front(e) is equivalent to: Q = {e, Q}
```

5.14.2.7 Push_back()

The prototype of the `push_back()` method is as follows:

```
function void push_back(input element_t item); );
```

The `push_back()` method inserts the given element at the end of the queue.

```
Q.push_back(e) is equivalent to: Q = {Q, e}
```

5.15 Array manipulation methods

SystemVerilog provides several built-in methods to facilitate array searching, ordering, and reduction.

The general syntax to call these array methods is as follows:

---

array_method_call ::=   *// not in Annex A*
     expression **.** array_method_name { attribute_instance } [**(** list_of_arguments **)**]
       [**with (** expression **)**]

Syntax 5-3—Array method call syntax (not in Annex A)

The optional **with** clause accepts an expression enclosed in parenthesis. In contrast, the **with** clause used by the randomize method (see 13.6) accepts a set of constraints enclosed in braces.

5.15.1 Array locator methods

Array locator methods operate on any unpacked array, including queues, but their return type is a queue. These locator methods allow searching an array for elements (or their indexes) that satisfy a given expression. Array locator methods traverse the array in an unspecified order. The optional **with** expression should not include any side effects; if it does, the results are unpredictable.

The prototype of these methods is as follows:

```
function array_type [$] locator_method (array_type iterator = item);
    // same type as the array
```

or

```
function int_or_index_type [$] index_locator_method(array_type iterator = item);
    // index type
```

Index locator methods return a queue of **int** for all arrays except associative arrays, which return a queue of the same type as the associative index type.

If no elements satisfy the given expression or the array is empty (in the case of a queue or dynamic array), then an empty queue is returned. Otherwise, these methods return a queue containing all items that satisfy the expression. Index locator methods return a queue with the indexes of all items that satisfy the expression. The optional expression specified by the **with** clause must evaluate to a boolean value.

Locator methods iterate over the array elements, which are then used to evaluate the expression specified by the **with** clause. The iterator argument optionally specifies the name of the variable used by the **with** expression to designate the element of the array at each iteration. If it is not specified, the name item is used by default. The scope for the iterator name is the **with** expression.

The following locator methods are supported (the **with** clause is mandatory):
— find() returns all the elements satisfying the given expression.
— find_index() returns the indexes of all the elements satisfying the given expression.
— find_first() returns the first element satisfying the given expression.
— find_first_index() returns the index of the first element satisfying the given expression.
— find_last() returns the last element satisfying the given expression.
— find_last_index() returns the index of the last element satisfying the given expression.

For the following locator methods, the **with** clause (and its expression) can be omitted if the relational operators (<, >, ==) are defined for the element type of the given array. If a **with** clause is specified, the relational operators (<, >, ==) must be defined for the type of the expression.
— min() returns the element with the minimum value or whose expression evaluates to a minimum.
— max() returns the element with the maximum value or whose expression evaluates to a maximum.
— **unique**() returns all elements with unique values or whose expression is unique.
— unique_index() returns the indexes of all elements with unique values or whose expression is unique.

Examples:

```
string SA[10], qs[$];
int IA[*], qi[$];

// Find all items greater than 5
qi = IA.find( x ) with ( x > 5 );

// Find indexes of all items equal to 3
qi = IA.find_index with ( item == 3 );

// Find first item equal to Bob
qs = SA.find_first with ( item == "Bob" );

// Find last item equal to Henry
qs = SA.find_last( y ) with ( y == "Henry" );

// Find index of last item greater than Z
qi = SA.find_last_index( s ) with ( s > "Z" );

// Find smallest item
qi = IA.min;

// Find string with largest numerical value
qs = SA.max with ( item.atoi );

// Find all unique strings elements
qs = SA.unique;

// Find all unique strings in lowercase
qs = SA.unique( s ) with ( s.tolower );
```

5.15.2 Array ordering methods

Array ordering methods can reorder the elements of one-dimensional arrays or queues.

The general prototype for the ordering methods is as follows:

```
function void ordering_method ( array_type iterator = item )
```

The following ordering methods are supported:
— reverse() reverses all the elements of the array (packed or unpacked). Specifying a **with** clause
  shall be a compiler error.
— sort() sorts the unpacked array in ascending order, optionally using the expression in the **with**
  clause. The **with** clause (and its expression) is optional when the relational operators are defined for
  the array element type.
— rsort() sorts the unpacked array in descending order, optionally using the expression in the **with**
  clause. The **with** clause (and its expression) is optional when the relational operators are defined for
  the array element type.
— shuffle() randomizes the order of the elements in the array. Specifying a **with** clause shall be a
  compiler error.

Examples:

```
string s[] = { "hello", "sad", "world" };
s.reverse;          // s becomes { "world", "sad", "hello" };
```

```
logic [4:1] b = 4'bXZ01;
b.reverse;          // b becomes 4'b10ZX

int q[$] = { 4, 5, 3, 1 };
q.sort;             // q becomes { 1, 3, 4, 5 }

struct { byte red, green, blue; } c [512];
c.sort with ( item.red );    // sort c using the red field only
c.sort( x ) with ( x.blue << 8 + x.green );  // sort by blue then green
```

5.15.3 Array reduction methods

Array reduction methods can be applied to any unpacked array to reduce the array to a single value. The expression within the optional **with** clause can be used to specify the item to use in the reduction.

The prototype for these methods is as follows:

```
function expression_or_array_type reduction_method (array_type iterator = item)
```

The method returns a single value of the same type as the array element type or, if specified, the type of the expression in the **with** clause. The **with** clause can be omitted if the corresponding arithmetic or boolean reduction operation is defined for the array element type. If a **with** clause is specified, the corresponding arithmetic or boolean reduction operation must be defined for the type of the expression.

The following reduction methods are supported:

— sum() returns the sum of all the array elements or, if a **with** clause is specified, returns the sum of the values yielded by evaluating the expression for each array element.

— product() returns the product of all the array elements or, if a **with** clause is specified, returns the product of the values yielded by evaluating the expression for each array element.

— **and**() returns the bitwise AND (&) of all the array elements or, if a **with** clause is specified, returns the bitwise AND of the values yielded by evaluating the expression for each array element.

— **or**() returns the bitwise OR (|) of all the array elements or, if a **with** clause is specified, returns the bitwise OR of the values yielded by evaluating the expression for each array element.

— **xor**() returns the logical XOR (^) of all the array elements or, if a **with** clause is specified, returns the XOR of the values yielded by evaluating the expression for each array element.

Examples:

```
byte b[] = { 1, 2, 3, 4 };
int y;

y = b.sum ;                    // y becomes 10 => 1 + 2 + 3 + 4
y = b.product ;                // y becomes 24 => 1 * 2 * 3 * 4
y = b.xor with ( item + 4 );   // y becomes 12 => 5 ^ 6 ^ 7 ^ 8
```

5.15.4 Iterator index querying

The expressions used by array manipulation methods sometimes need the actual array indexes at each iteration, not just the array element. The index method of an iterator returns the index value of the specified dimension. The prototype of the index method is as follows:

```
function int_or_index_type index ( int dimension = 1 )
```

The array dimensions are numbered as defined in 22.6: The slowest varying is dimension 1. Successively faster varying dimensions have sequentially higher dimension numbers. If the dimension is not specified, the first dimension is used by default

The return type of the index method is an **int** for all array iterator items except associative arrays, which return an index of the same type as the associative index type.

For example:

```
int arr[]
int mem[9:0][9:0], mem2[9:0][9:0];
int q[$];
...

// find all items equal to their position (index)
q = arr.find with ( item == item.index );

// find all items in mem that are greater than corresponding item in mem2
q = mem.find( x ) with ( x > mem2[x.index(1)][x.index(2)] );
```

6. Data declarations

6.1 Introduction

NOTE—There are several forms of data in SystemVerilog: literals (see Clause 3), parameters, constants, variables, nets, and attributes (see 4.17). A data object is a named entity that has a data value associated with it, such as a parameter, a variable, or a net.

Verilog constants are literals, genvars parameters, localparams, and specparams. Verilog also has variables and nets. Variables must be written by procedural statements, and nets must be written by continuous assignments or ports.

SystemVerilog extends the functionality of variables by allowing them to be either written by procedural statements or driven by a single continuous assignment, similar to a **wire**. Because the keyword **reg** no longer describes the user's intent in many cases, the keyword **logic** is added as a more accurate description that is equivalent to **reg**. See 6.9.2 for details on SystemVerilog type equivalence rules. Verilog has already deprecated the use of the term *register* in favor of *variable*.

SystemVerilog follows Verilog by requiring data to be declared before they are used, apart from implicit nets. The rules for implicit nets are the same as in Verilog.

A variable can be static (storage allocated on instantiation and never deallocated) or automatic (stack storage allocated on entry to a scope, such as a task, function, or block, and deallocated on exit). C has the keywords **static** and **auto**. SystemVerilog follows Verilog in respect of the static default storage class, with automatic tasks and functions, but allows **static** to override a default of **automatic** for a particular variable in such tasks and functions.

SystemVerilog extends the set of data types that are available for modeling Verilog storage and transmission elements. In addition to the Verilog data types, new predefined data types and user-defined data types can be used to declare constants, variables, and nets.

6.2 Data declaration syntax

data_declaration[14] ::=   *// from A.2.1.3*
      [**const**] [lifetime] data_type_or_implicit list_of_variable_decl_assignments **;**
    | type_declaration
    | package_import_declaration
    | virtual_interface_declaration

net_declaration[13] ::=
      net_type [drive_strength | charge_strength] [**vectored** | **scalared**]
        data_type_or_implicit [delay3] list_of_net_decl_assignments **;**

lifetime ::= **static** | **automatic**

13. A charge strength shall only be used with the **trireg** keyword. When the **vectored** or **scalared** keyword is used, there shall be at least one packed dimension.

14. In a data_declaration that is not within the procedural context, it shall be illegal to use the **automatic** keyword. In a data_declaration, it shall be illegal to omit the explicit data_type before a list_of_variable_decl_assignments unless the **var** keyword is used.

Syntax 6-1—Data declaration syntax (excerpt from Annex A)

6.3 Constants

Constants are named data variables that never change. Verilog provides three constructs for defining elaboration-time constants: the **parameter**, **localparam** and **specparam** declarations.

All three can be initialized with a literal.

```
localparam byte colon1 = ":" ;
specparam delay = 10 ; // specparams are used for specify blocks
const logic flag = 1 ;
```

Verilog provides four methods for setting the value of parameter constants in a design. Each parameter must be assigned a default value when declared. The default value of a parameter of an instantiated module can be overridden in each instance of the module using one of the following:

— Implicit in-line parameter redefinition (e.g., `foo #(value, value) u1 (...); )`

— Explicit in-line parameter redefinition (e.g., `foo #(.name(value), .name(value)) u1 (...); )`

— **defparam** statements, using hierarchical path names to redefine each parameter

NOTE—The **defparam** statement might be removed from future versions of the language. See 25.2.

6.3.1 Parameter declaration syntax

local_parameter_declaration ::=	// from *A.2.1.1*
localparam data_type_or_implicit list_of_param_assignments **;**	
| **localparam type** list_of_type_assignments **;**	
parameter_declaration ::=	
parameter data_type_or_implicit list_of_param_assignments	
| **parameter type** list_of_type_assignments	
specparam_declaration ::=	
specparam [packed_dimension] list_of_specparam_assignments **;**	
data_type_or_implicit ::=	// from A.2.2.1
data_type	
| [signing] { packed_dimension }	
type_reference ::=	
type (expression[27] **)**	
| **type (** data_type **)**	
list_of_param_assignments ::= param_assignment { **,** param_assignment }	// from *A.2.3*
list_of_specparam_assignments ::= specparam_assignment { **,** specparam_assignment }	
list_of_type_assignments ::= type_assignment { **,** type_assignment }	
param_assignment ::=	// from *A.2.4*
parameter_identifier { unpacked_dimension } **=** constant_param_expression	
specparam_assignment ::=	
specparam_identifier **=** constant_mintypmax_expression	
| pulse_control_specparam	
type_assignment ::=	
type_identifier **=** data_type	

Syntax 6-2—Parameter declaration syntax (excerpt from Annex A)

```
parameter_port_list ::=                                                          // from A.1.3
      # ( list_of_param_assignments { , parameter_port_declaration } )
      | # ( parameter_port_declaration { , parameter_port_declaration } )
      | #( )
parameter_port_declaration ::=
      parameter_declaration
      | data_type list_of_param_assignments
      | type list_of_type_assignments
```

27. An expression that is used as the argument in a type_reference shall not contain any hierarchical references or references to elements of dynamic objects.

Syntax 6-2—Parameter declaration syntax (excerpt from Annex A) (continued)

6.3.2 Value parameters

A module, interface, program, or class can have parameters, which are set during elaboration and are constant during simulation. They are defined with data types and default values. For compatibility with Verilog, if no data type is supplied, the type is determined when the value is determined.

In an assignment to, or override of, a parameter without an explicit type declaration, the type of the right-hand expression shall be real or integral. If the expression is real, the parameter is real. If the expression is integral, the parameter is a `logic` vector of the same size with range `[size-1:0]`. In an assignment to, or override of, a parameter with an explicit type declaration, the type of the right-hand expression shall be assignment compatible with the declared type.

Unlike nonlocal parameters, local parameters can be declared in a `generate` block, in a package, or in a compilation-unit scope. In these contexts, the `parameter` keyword can be used as a synonym for the `localparam` keyword.

6.3.2.1 $ as a parameter value

The value **$** can be assigned to parameters of integer types. A parameter to which **$** is assigned shall only be used wherever **$** can be specified as a literal constant.

For example, $ represents unbounded range specification, where the upper index can be any integer.

```
parameter r2 = $;
property inq1(r1,r2);
    @(posedge clk) a ##[r1:r2] b ##1 c |=> d;
endproperty
assert inq1(3);
```

To support whether a constant is **$**, a system function is provided to test whether a constant is a **$**. The syntax of the system function is

```
$isunbounded(const_expression);
```

`$isunbounded` returns true if *const_expression* is unbounded. Typically, `$isunbounded` would be used as a condition in the generate statement.

The example below illustrates the benefit of using $ in writing properties concisely where the range is parameterized. The checker in the example ensures that a bus driven by signal en remains 0, i.e, quiet for the specified minimum (`min_quiet`) and maximum (`max_quiet`) quiet time.

NOTE—The function `$isunbounded` is used for checking the validity of the actual arguments.

```
interface quiet_time_checker #(parameter min_quiet = 0,
                               parameter max_quiet = 0)
                              (input logic clk, reset_n, logic [1:0]en);

    generate
        if ( max_quiet == 0) begin
            property quiet_time;
                @(posedge clk) reset_n |-> ($countones(en) == 1);
            endproperty
            a1: assert property (quiet_time);
        end
        else begin
            property quiet_time;
                @(posedge clk)
                    (reset_n && ($past(en) != 0) && en == 0)
                    |->(en == 0)[*min_quiet:max_quiet]
                ##1 ($countones(en) == 1);
            endproperty
            a1: assert property (quiet_time);
        end
        if ((min_quiet == 0) && ($isunbounded(max_quiet))
            $display(warning_msg);
    endgenerate
endinterface


quiet_time_checker #(0, 0) quiet_never (clk,1,enables);
quiet_time_checker #(2, 4) quiet_in_window (clk,1,enables);
quiet_time_checker #(0, $) quiet_any (clk,1,enables);
```

Another example below illustrates that by testing for **$**, a property can be configured according to the requirements. When parameter `max_cks` is unbounded, it is not required to test for expr to become false.

```
interface width_checker #(parameter min_cks = 1, parameter max_cks = 1)
                          (input logic clk, reset_n, expr);

    generate
        if ($isunbounded(max_cks)) begin
            property width;
                @(posedge clk)
                    (reset_n && $rose(expr)) |-> (expr [* min_cks]);
            endproperty
            a2: assert property (width);
        end
        else begin
            property assert_width_p;
                @(posedge clk)
                    (reset_n && $rose(expr)) |-> (expr[* min_cks:max_cks])
                        ##1 (!expr);
            endproperty
            a2: assert property (width);
        end
```

```
        endgenerate
    endinterface


    width_checker #(3, $) max_width_unspecified (clk,1,enables);
    width_checker #(2, 4) width_specified (clk,1,enables);
```

6.3.3 Type parameters

SystemVerilog adds the ability for a parameter to also specify a data type, allowing modules or instances to have data whose type is set for each instance.

```
    module ma   #( parameter p1 = 1, parameter type p2 = shortint )
                (input logic [p1:0] i, output logic [p1:0] o);
       p2 j = 0; // type of j is set by a parameter, (shortint unless redefined)
       always @(i) begin
          o = i;
          j++;
       end
    endmodule

    module mb;
       logic [3:0] i,o;
       ma #(.p1(3), .p2(int)) u1(i,o); //redefines p2 to a type of int
    endmodule
```

In an assignment to, or override of, a type parameter, the right-hand expression shall represent a data type.

It is an error to override a type parameter with a **defparam** statement.

6.3.4 Parameter port lists

SystemVerilog also adds the ability to omit the **parameter** keyword in a parameter port list.

```
    class vector #(size = 1);
       logic [size-1:0] v;
    endclass

    typedef vector#(16) word;

    interface simple_bus #(AWIDTH = 64, type T = word) (input bit clk) ;
    endinterface
```

In a list of parameters, a parameter can depend on earlier parameters. In the following declaration, the default value of the second parameter depends on the value of the first parameter. The third parameter is a type, and the fourth parameter is a value of that type.

```
    module mc # (int N = 5, M = N*16, type T = int, T x = 0)
      ( ... );
    ...
    endmodule
```

6.3.5 Const constants

SystemVerilog adds another form of a local constant, **const**. A **const** form of constant differs from a **localparam** constant in that the **localparam** must be set during elaboration, whereas a **const** can be set during simulation, such as in an automatic task.

A value parameter (**parameter**, **localparam**, or **specparam**) can only be set to an expression of literals, value parameters or local parameters, genvars, enumerated names, or a constant function of these. Package references are allowed. Hierarchical names are not allowed. A **specparam** can also be set to an expression containing one or more specparams.

A data-type parameter (**parameter type**) can only be set to a data type. Package references are allowed. Hierarchical names are not allowed.

A static constant declared with the **const** keyword can be set to an expression of literals, parameters, local parameters, genvars, enumerated names, a constant function of these, or other constants. Hierarchical names are allowed because constants declared with the **const** keyword are calculated after elaboration.

```
const logic option = a.b.c ;
```

An automatic constant declared with the **const** keyword can be set to any expression that would be legal without the **const** keyword.

An instance of a class (an object handle) can also be declared with the **const** keyword.

```
const class_name object = new(5,3);
```

In other words, the object acts like a variable that cannot be written. The arguments to the **new** method must be constant expressions. The members of the object can be written (except for those members that are declared **const**).

6.4 Variables

One form of variable declaration consists of a data type followed by one or more instances.

```
shortint s1, s2[0:9];
```

Another form of variable declaration begins with the keyword **var**. The data type is optional in this case. If a data type is not specified, then the data type **logic** shall be inferred.

```
var byte my_byte;     // equivalent to "byte my_byte;"
var v;                // equivalent to "var logic v;"
var [15:0] vw;        // equivalent to "var logic [15:0] vw;"
var enum bit { clear, error } status;
input var logic data_in;
var reg r;
```

A variable can be declared with an initializer, for example:

```
int i = 0;
```

In Verilog, an initialization value specified as part of the declaration is executed as if the assignment were made from an **initial** block, after simulation has started. In SystemVerilog, setting the initial value of a static variable as part of the variable declaration (including static class members) shall occur before any **initial** or **always** blocks are started.

Initial values in SystemVerilog are not constrained to simple constants; they can include run-time expressions, including dynamic memory allocation. For example, a static class handle or a mailbox can be created and initialized by calling its **new** method (see 14.3.1), or static variables can be initialized to random values by calling the $urandom system task. This requires a special pre-initial pass at run time.

Table 6-1 contains the default values for SystemVerilog variables.

Table 6-1—Default values

Type	Default initial value
4-state integral	`'X`
2-state integral	`'0`
real, shortreal	`0.0`
Enumeration	base type default initial value
string	`""` (empty string)
event	New event
class	**null**
chandle (Opaque handle)	**null**

6.5 Nets

A net declaration begins with a net type that determines how the values of the nets in the declaration are resolved. The declaration can include optional information such as delay values and drive or charge strength.

Verilog restricts the data type of a net to a scalar, a bit vector, or an array of scalars or bit vectors. In System-Verilog, any 4-state data type can be used to declare a net. For example:

```
trireg (large) logic #(0,0,0) cap1;
typedef logic [31:0] addressT;
wire addressT w1;
wire struct packed { logic ecc; logic [7:0] data; } memsig;
```

If a data type is not specified in the net declaration, then the data type of the net is **logic**.

Certain restrictions apply to the data type of a net. A valid data type for a net shall be one of the following:

    a)   A 4-state integral type, including a packed array or packed **struct**

    b)   An unpacked array or unpacked **struct**, where each element has a valid data type for a net

The effect of this recursive definition is that a net is composed entirely of 4-state bits and is treated accordingly. There is no change to the Verilog network semantics. In addition to a signal value, each bit of a net shall have additional strength information. When bits of signals combine, the strength and value of the resulting signal shall be determined as in 7.10 of IEEE Std 1364.

There is no change in the treatment of the signed property across hierarchical boundaries.

A lexical restriction applies to the use of the **reg** keyword in a net or port declaration. A Verilog net type keyword shall not be followed directly by the **reg** keyword. Thus, the following declarations are in error:

```
tri reg r;
inout wire reg p;
```

The **reg** keyword can be used in a net or port declaration if there are lexical elements between the net type keyword and the **reg** keyword.

6.6 Scope and lifetime

Any data declared outside a module, interface, task, or function are global in scope (can be used anywhere after its declaration) and have a static lifetime (exist for the whole elaboration and simulation time).

SystemVerilog data declared inside a module or interface, but outside a task, process, or function, are local in scope and static in lifetime (exist for the lifetime of the module or interface). This is roughly equivalent to C static data declared outside a function, which is local to a file.

Data declared in an automatic task, function, or block have the lifetime of the call or activation and a local scope. This is roughly equivalent to a C automatic variable.

Data declared in a static task, function, or block default to a static lifetime and a local scope.

In SystemVerilog, data can be declared in unnamed blocks as well as in named blocks. These data are visible to the unnamed block and any nested blocks below it. Hierarchical references cannot be used to access these data by name.

Verilog allows tasks and functions to be declared as **automatic**, making all storage within the task or function automatic. SystemVerilog allows specific data within a static task or function to be explicitly declared as **automatic**. Data declared as automatic have the lifetime of the call or block and are initialized on each entry to the call or block. The lifetime of a **fork**…**join**, **fork**…**join_any**, or **fork**…**join_none** block shall encompass the execution of all processes spawned by the block. The lifetime of a scope enclosing any **fork** block includes the lifetime of the **fork** block.

SystemVerilog also allows data to be explicitly declared as **static**. Data declared to be **static** in an automatic task, function, or block have a static lifetime and a scope local to the block. This is like C static data declared within a function.

```
module ms1;
    int st0; // static
    initial begin
        int st1; //static
        static int st2; //static
        automatic int auto1; //automatic
    end
    task automatic t1();
        int auto2; //automatic
        static int st3; //static
        automatic int auto3; //automatic
    endtask
endmodule
```

SystemVerilog adds an optional qualifier to specify the default lifetime of all variables declared in a task, function, or block defined within a module, interface, or program (see Clause 16). The lifetime qualifier is **automatic** or **static**. The default lifetime is **static**.

```
program automatic test ;
    int i;                  // not within a procedural block - static
    task foo( int a );      // arguments and variables in foo are automatic
        ...                 // unless explicitly declared static
    endtask
```

```
    endprogram
```

It is permissible to hierarchically reference any static variable unless the variable is declared inside an unnamed block. This includes static variables declared inside automatic tasks and functions.

Class methods and declared `for` loop variables are by default automatic, regardless of the lifetime attribute of the scope in which they are declared. Classes are discussed in Clause 7.

Automatic variables and members or elements of dynamic variables—class properties and dynamically sized variables—shall not be written with nonblocking, continuous, or procedural continuous assignments. References to automatic variables and elements or members of dynamic variables shall be limited to procedural blocks.

See also Clause 12 on tasks and functions.

6.7 Nets, regs, and logic

Verilog states that a net can be written by one or more continuous assignments, by primitive outputs, or through module ports. The resultant value of multiple drivers is determined by the resolution function of the net type. A net cannot be procedurally assigned. If a net on one side of a port is driven by a variable on the other side, a continuous assignment is implied. A `force` statement can override the value of a net. When released, it returns to resolved value.

Verilog also states that one or more procedural statements can write to variables, including procedural continuous assignments. The last write determines the value. A variable cannot be continuously assigned. The `force` statement overrides the procedural `assign` statement, which in turn overrides the normal assignments. A variable cannot be written through a port; it must go through an implicit continuous assignment to a net.

In SystemVerilog, all variables can now be written either by one continuous assignment, or by one or more procedural statements, including procedural continuous assignments. It shall be an error to have multiple continuous assignments or a mixture of procedural and continuous assignments writing to any term in the expansion of a written longest static prefix of a logic variable (See 8.11 for the definition of the expansion of a longest static prefix). All data types can write through a port.

SystemVerilog variables can be packed or unpacked aggregates of other types. Multiple assignments made to independent elements of a variable are examined individually. An assignment where the left-hand side contains a slice is treated as a single assignment to the entire slice. It shall be an error to have a packed structure or array type written with a mixture of procedural and continuous assignments. Thus, an unpacked structure or array can have one element assigned procedurally and another element assigned continuously. And, elements of a packed structure or array can be assigned with multiple continuous assignments, provided that each bit is covered by no more than a single continuous assignment. For example, assume the following structure declaration:

```
    struct {
        bit [7:0] A;
        bit [7:0] B;
        byte C;
    } abc;
```

The following statements are legal assignments to `struct` abc:

```
    assign abc.C = sel ? 8'hBE : 8'hEF;
```

```
not    (abc.A[0],abc.B[0]),
       (abc.A[1],abc.B[1]),
       (abc.A[2],abc.B[2]),
       (abc.A[3],abc.B[3]);

always @(posedge clk) abc.B <= abc.B + 1;
```

The following additional statements are illegal assignments to **struct** abc:

```
// Multiple continuous assignments to abc.C
assign abc.C = sel ? 8'hDE : 8'hED;

// Mixing continuous and procedural assignments to abc.A
always @(posedge clk) abc.A[7:4] <= !abc.B[7:4];
```

For the purposes of the preceding rule, a declared variable initialization or a procedural continuous assignment is considered a procedural assignment. A **force** statement is neither a continuous nor a procedural assignment. A **release** statement shall not change the variable until there is another procedural assignment or shall schedule a reevaluation of the continuous assignment driving it. A single **force** or **release** statement shall not be applied to a whole or part of a variable that is being assigned by a mixture of continuous and procedural assignments.

A continuous assignment is implied when a variable is connected to an input port declaration. This makes assignments to a variable declared as an input port illegal. A continuous assignment is implied when a variable is connected to the output port of an instance. This makes additional procedural or continuous assignments to a variable connected to the output port of an instance illegal.

SystemVerilog variables cannot be connected to either side of an inout port. SystemVerilog introduces the concept of shared variables across ports with the **ref** port type. See 19.12 for more information about ports and port connection rules.

The compiler can issue a warning if a continuous assignment could drive strengths other than St0, St1, StX, or HiZ to a variable. In any case, SystemVerilog applies automatic type conversion to the assignment, and the strength is lost.

Unlike SystemVerilog nets, a SystemVerilog variable cannot have an implicit continuous assignment as part of its declaration, the way a net can. An assignment as part of the logic declaration is a variable initialization, not a continuous assignment. For example:

```
wire w = vara & varb;              // continuous assignment

logic v = consta & constb;         // initial procedural assignment

logic vw; // no initial assignment
assign vw = vara & varb;           // continuous assignment to a logic

real circ;
assign circ = 2.0 * PI * R;        // continuous assignment to a real
```

6.8 Signal aliasing

The Verilog **assign** statement is a unidirectional assignment and can incorporate a delay and strength change. To model a bidirectional short-circuit connection, it is necessary to use the **alias** statement. The members of an alias list are signals whose bits share the same physical nets. The example below implements a byte order swapping between bus A and bus B.

```
module byte_swap (inout wire [31:0] A, inout wire [31:0] B);
    alias {A[7:0],A[15:8],A[23:16],A[31:24]} = B;
endmodule
```

This example strips out the LSB and MSB from a 4-byte bus:

```
module byte_rip (inout wire [31:0] W, inout wire [7:0] LSB, MSB);
    alias W[7:0] = LSB;
    alias W[31:24] = MSB;
endmodule
```

The bit overlay rules are the same as for a packed union with the same member types: each member shall be the same size, and connectivity is independent of the simulation host. The nets connected with an **alias** statement must be type compatible, that is, they have to be of the same net type. For example, it is illegal to connect a **wand** net to a **wor** net with an **alias** statement. This rule is stricter than the rule applied to nets joining at ports because the scope of an alias is limited and such connections are more likely to be a design error. Variables and hierarchical references cannot be used in **alias** statements. Any violation of these rules shall be considered a fatal error.

The same nets can appear in multiple **alias** statements. The effects are cumulative. The following two examples are equivalent. In either case, low12[11:4] and high12[7:0] share the same wires.

```
module overlap(inout wire [15:0] bus16, inout wire [11:0] low12, high12);
    alias bus16[11:0] = low12;
    alias bus16[15:4] = high12;
endmodule
```

```
module overlap(inout wire [15:0] bus16, inout wire [11:0] low12, high12);
    alias bus16 = {high12, low12[3:0]};
    alias high12[7:0] = low12[11:4];
endmodule
```

To avoid errors in specification, it is not allowed to specify an alias from an individual signal to itself or to specify a given alias more than once. The following version of the code above would be illegal because the top 4 bits and bottom 4 bits are the same in both statements:

```
alias bus16 = {high12[11:8], low12};
alias bus16 = {high12, low12[3:0]};
```

This alternative is also illegal because the bits of bus16 are being aliased to itself:

```
alias bus16 = {high12, bus16[3:0]} = {bus16[15:12], low12};
```

alias statements can appear anywhere module instance statements can appear. If an identifier that has not been declared as a data type appears in an **alias** statement, then an implicit net is assumed, following the same rules as implicit nets for a module instance. The following example uses **alias** along with the automatic name binding to connect pins on cells from different libraries to create a standard macro:

```
module lib1_dff(Reset, Clk, Data, Q, Q_Bar);
    ...
endmodule
```

```
module lib2_dff(reset, clock, data, a, qbar);
    ...
endmodule
```

```
    module lib3_dff(RST, CLK, D, Q, Q_);
       ...
    endmodule

    macromodule my_dff(rst, clk, d, q, q_bar); // wrapper cell
       input rst, clk, d;
       output q, q_bar;
       alias rst = Reset = reset = RST;
       alias clk = Clk = clock = CLK;
       alias d = data = D;
       alias q = Q;
       alias Q_ = q_bar = Q_Bar = qbar;
       `LIB_DFF my_dff (.*); // LIB_DFF is any of lib1_dff, lib2_dff or lib3_dff
    endmodule
```

Using a net in an **alias** statement does not modify its syntactic behavior in other statements. Aliasing is performed at elaboration time and cannot be undone.

6.9 Type compatibility

Some SystemVerilog constructs and operations require a certain level of type compatibility for their operands to be legal. There are five levels of type compatibility, formally defined here: matching, equivalent, assignment compatible, cast compatible, and nonequivalent.

SystemVerilog does not require a category for identical types to be defined here because there is no construct in the SystemVerilog language that requires it. For example, as defined below, **int** can be interchanged with **bit signed** [31:0] wherever it is syntactically legal to do so. Users can define their own level of type identity by using the $typename system function (see 22.2) or through use of the PLI.

The scope of a data type identifier shall include the hierarchical instance scope. In other words, each instance with a user-defined type declared inside the instance creates a unique type. To have type matching or equivalence among multiple instances of the same module, interface, or program, a class, **enum**, unpacked structure, or unpacked union type must be declared at a higher level in the compilation-unit scope than the declaration of the module, interface, or program, or imported from a package. For type matching, this is true even for packed structure and packed union types.

6.9.1 Matching types

Two data types shall be defined as matching data types using the following inductive definition. If two data types do not match using the following definition, then they shall be defined to be nonmatching.

a)   Any built-in type matches every other occurrence of itself, in every scope.

b)   A simple **typedef** or type parameter override that renames a built-in or user-defined type matches that built-in or user-defined type within the scope of the type identifier.

```
    typedef bit node;        // 'bit' and 'node' are matching types
    typedef type1 type2;     // 'type1' and 'type2' are matching types
```

c)   An anonymous **enum**, **struct**, or **union** type matches itself among data objects declared within the same declaration statement and no other data types.

```
    struct packed {int A; int B;} AB1, AB2; // AB1, AB2 have matching types
    struct packed {int A; int B;} AB3;     // the type of AB3 does not match the
                                           // type of AB1
```

d) A **typedef** for an **enum**, **struct**, **union**, or **class** matches itself and the type of data objects declared using that data type within the scope of the data type identifier.

```
typedef struct packed {int A; int B;} AB_t;
AB_t AB1; AB_t AB2;  // AB1 and AB2 have matching types

typedef struct packed {int A; int B;} otherAB_t;
otherAB_t AB3;       // the type of AB3 does not match the type of AB1 or AB2
```

e) A simple bit vector type that does not have a predefined width and one that does have a predefined width match if both are 2-state or both are 4-state, both are signed or both are unsigned, both have the same width, and the range of the simple bit vector type without a predefined width is [width–1:0].

```
typedef bit signed [7:0] BYTE;   // matches the byte type
typedef bit signed [0:7] ETYB;   // does not match the byte type
```

f) Two array types match if they have the same number of unpacked dimensions and their slowest varying dimensions have matching types and the same left and right range bounds. The type of the slowest varying dimension of a multidimensional array type is itself an array type.

```
typedef byte MEM_BYTES [256];
typedef bit signed [7:0] MY_MEM_BYTES [256];  // MY_MEM_BYTES matches
MEM_BYTES

typedef logic [1:0] [3:0] NIBBLES;
typedef logic [7:0] MY_BYTE;  // MY_BYTE and NIBBLES are not matching types
```

g) Explicitly adding **signed** or **unsigned** modifiers to a type that does not change its default signing creates a type that matches the type without the explicit signing specification.

```
typedef byte signed MY_CHAR;  // MY_CHAR matches the byte type
```

h) A **typedef** for an **enum**, **struct**, **union**, or **class** type declared in a package always matches itself, regardless of the scope into which the type is imported.

6.9.2 Equivalent types

Two data types shall be defined as equivalent data types using the following inductive definition. If the two data types are not defined equivalent using the following definition, then they shall be defined to be nonequivalent.

a) If two types match, they are equivalent.

b) An anonymous **enum**, unpacked **struct**, or unpacked **union** type is equivalent to itself among data objects declared within the same declaration statement and no other data types.

```
struct {int A; int B;} AB1, AB2;     // AB1, AB2 have equivalent types
struct {int A; int B;} AB3;          // AB3 is not type equivalent to AB1
```

c) Packed arrays, packed structures, packed unions, and built-in integral types are equivalent if they contain the same number of total bits, are either all 2-state or all 4-state, and are either all signed or all unsigned.

NOTE—If any bit of a packed structure or union is 4-state, the entire structure or union is considered 4-state.

```
        typedef bit signed [7:0] BYTE;      // equivalent to the byte type
        typedef struct packed signed {bit[3:0] a, b;} uint8;
                                             // equivalent to the byte type
```

d) Unpacked array types are equivalent by having equivalent element types and identical shape. Shape is defined as the number of dimensions and the number of elements in each dimension, not the actual range of the dimension.

```
        bit [9:0]  A [0:5];
        bit [1:10] B [6];
        typedef bit [10:1] uint10;
        uint10 C [6:1];              // A, B and C have equivalent types
        typedef int anint [0:0];     // anint is not type equivalent to int
```

The following example is assumed to be within one compilation unit, although the package declaration need not be in the same unit:

```
    package p1;
        typedef struct {int A;} t_1;
    endpackage

    typedef struct {int A;} t_2;

    module sub();
        import p1::t_1;
        parameter type t_3 = int;
        parameter type t_4 = int;
        typedef struct {int A;} t_5;
        t_1 v1; t_2 v2; t_3 v3; t_4 v4; t_5 v5;
    endmodule

    module top();
        typedef struct {int A;} t_6;
        sub #(.t_3(t_6)) s1 ();
        sub #(.t_3(t_6)) s2 ();

        initial begin
            s1.v1 = s2.v1; // legal - both types from package p1 (rule 8)
            s1.v2 = s2.v2; // legal - both types from $unit (rule 4)
            s1.v3 = s2.v3; // legal - both types from top (rule 2)
            s1.v4 = s2.v4; // legal - both types are int (rule 1)
            s1.v5 = s2.v5; // illegal - types from s1 and s2 (rule 4)
        end
    endmodule
```

6.9.3 Assignment compatible

All equivalent types, and all nonequivalent types that have implicit casting rules defined between them, are assignment-compatible types. For example, all integral types are assignment compatible. Conversion between assignment-compatible types can involve loss of data by truncation or rounding.

Compatibility can be in one direction only. For example, an **enum** can be converted to an integral type without a cast, but not the other way around. Implicit casting rules are defined in Clause 4 and Clause 8.

6.9.4 Cast compatible

All assignment-compatible types, plus all nonequivalent types that have defined explicit casting rules, are cast -compatible types. For example, an integral type requires a cast to be assigned to an **enum**.

Explicit casting rules are defined in Clause 4.

6.9.5 Type incompatible

Type incompatible includes all the remaining nonequivalent types that have no defined implicit or explicit casting rules. Class handles and chandles are type incompatible with all other types.

6.10 Type operator

The **type** operator provides a way to refer to the data type of an expression. A type reference can be used like a type name or local type parameter, for example, in casts, data object declarations, and type parameter assignments and overrides. It can also be used in equality/inequality and case equality/inequality comparisons with other type references, and such comparisons are considered to be constant expressions. When a type reference is used in a net declaration, it shall be preceded by a net **type** keyword; and when it is used in a variable declaration, it shall be preceded by the **var** keyword.

```
var type(a+b) c, d;

c = type(i+3)'(v[15:0]);
```

The **type** operator applied to an expression shall represent the self-determined result type of that expression. The expression shall not be evaluated and shall not contain any hierarchical references or references to elements of dynamic objects.

The type operator can also be applied to a data type.

```
localparam type T = type(bit[12:0]);
```

When a type reference is used in an equality/inequality or case equality/inequality comparison, it shall only be compared with another type reference. Two type references shall be considered equal in such comparisons if, and only if, the types to which they refer match (see 6.9.1).

```
bit [12:0] A_bus, B_bus;
parameter type bus_t = type(A_bus);
generate
  case (type(bus_t))
    type(bit[12:0]): addfixed_int #(bus_t) (A_bus,B_bus);
    type(real): add_float #(type(A_bus)) (A_bus,B_bus);
  endcase
endgenerate
```

7. Classes

7.1 Introduction

NOTE—SystemVerilog introduces an object-oriented **class** data abstraction. Classes allow objects to be dynamically created, deleted, assigned, and accessed via object handles. Object handles provide a safe pointer-like mechanism to the language. Classes offer inheritance and abstract type modeling, which brings the advantages of C function pointers with none of the type-safety problems and, thus, brings true polymorphism into Verilog.

7.2 Syntax

```
class_declaration ::=                                                         // from A.1.2
        [ virtual ] class [ lifetime ] class_identifier [ parameter_port_list ]
            [ extends class_type [ ( list_of_arguments ) ] ];
            { class_item }
        endclass [ : class_identifier]
class_item ::=                                                                 // from A.1.8
        { attribute_instance } class_property
      | { attribute_instance } class_method
      | { attribute_instance } class_constraint
      | { attribute_instance } class_declaration
      | { attribute_instance } timeunits_declaration17
      | { attribute_instance } covergroup_declaration
      | ;
class_property ::=
        { property_qualifier } data_declaration
      | const { class_item_qualifier } data_type const_identifier [ = constant_expression ] ;
class_method ::=
        { method_qualifier } task_declaration
      | { method_qualifier } function_declaration
      | extern { method_qualifier } method_prototype ;
      | { method_qualifier } class_constructor_declaration
      | extern { method_qualifier } class_constructor_prototype
class_constructor_prototype ::=
        function new ( [ tf_port_list ] ) ;
class_constraint ::=
        constraint_prototype
      | constraint_declaration
class_item_qualifier7 ::=
        static
      | protected
      | local
property_qualifier7 ::=
        random_qualifier
      | class_item_qualifier
random_qualifier7 ::=
        rand
      | randc
```

Syntax 7-1—Class syntax (excerpt from Annex A)

```
method_qualifier⁷ ::=
        virtual
      | class_item_qualifier
method_prototype ::=
        task_prototype
      | function_prototype
class_constructor_declaration ::=
        function [ class_scope ] new [ ( [ tf_port_list ] ) ] ;
            { block_item_declaration }
            [ super . new [ ( list_of_arguments ) ] ; ]
            { function_statement_or_null }
        endfunction [ : new ]
```

Syntax 7-1—Class syntax (excerpt from [Annex A](#)) (continued)

7.3 Overview

A class is a type that includes data and subroutines (functions and tasks) that operate on those data. A class's data are referred to as *class properties*, and its subroutines are called *methods*; both are members of the class. The class properties and methods, taken together, define the contents and capabilities of some kind of object.

For example, a packet might be an object. It might have a command field, an address, a sequence number, a time stamp, and a packet payload. In addition, there are various things than can be done with a packet: initialize the packet, set the command, read the packet's status, or check the sequence number. Each packet is different; but as a class, packets have certain intrinsic properties that can be captured in a definition.

```
class Packet ;
   //data or class properties
   bit [3:0] command;
   bit [40:0] address;
   bit [4:0] master_id;
   integer time_requested;
   integer time_issued;
   integer status;

   // initialization
   function new();
      command = IDLE;
      address = 41'b0;
      master_id = 5'bx;
   endfunction

   // methods
   // public access entry points
   task clean();
      command = 0; address = 0; master_id = 5'bx;
   endtask

   task issue_request( int delay );
      // send request to bus
   endtask

   function integer current_status();
      current_status = status;
   endfunction
```

```
    endclass
```

A common convention is to capitalize the first letter of the class name so that it is easy to recognize class declarations.

7.4 Objects (class instance)

A class defines a data type. An object is an instance of that class. An object is used by first declaring a variable of that class type (that holds an object handle) and then creating an object of that class (using the **new** function) and assigning it to the variable.

```
    Packet p; // declare a variable of class Packet
    p = new;  // initialize variable to a new allocated object of the class Packet
```

The variable `p` is said to hold an object handle to an object of class `Packet`.

Uninitialized object handles are set by default to the special value **null.** An uninitialized object can be detected by comparing its handle with **null**.

For example: The task `task1` below checks whether the object is initialized. If it is not, it creates a new object via the **new** command.

```
    class obj_example;
          ...
    endclass

    task task1(integer a, obj_example myexample);
       if (myexample == null) myexample = new;
    endtask
```

Accessing nonstatic members (see 7.8) or virtual methods (see 7.19) via a **null** object handle is illegal. The result of an illegal access via a null object is indeterminate, and implementations can issue an error.

SystemVerilog objects are referenced using an object handle. There are some differences between a C pointer and a SystemVerilog object handle (see Table 7-1). C pointers give programmers a lot of latitude in how a pointer can be used. The rules governing the usage of SystemVerilog object handles are much more restrictive. A C pointer can be incremented, for example; but a SystemVerilog object handle cannot. In addition to object handles, 4.6 introduces the **chandle** data type for use with the DPI (see Clause 26).

7.5 Object properties

The data fields of an object can be used by qualifying class property names with an instance name. Using the earlier example, the commands for the `Packet` object `p` can be used as follows:

```
    Packet p = new;
    p.command = INIT;
    p.address = $random;
    packet_time = p.time_requested;
```

There are no restrictions on the data type of a class property.

Table 7-1—Comparison of pointer and handle types

Operation	C pointer	SV object handle	SV chandle
Arithmetic operations (such as incrementing)	Allowed	Not allowed	Not allowed
For arbitrary data types	Allowed	Not allowed	Not allowed
Dereference when null	Error	Not allowed	Not allowed
Casting	Allowed	Limited	Not allowed
Assignment to an address of a data type	Allowed	Not allowed	Not allowed
Unreferenced objects are garbage collected	No	Yes	No
Default value	Undefined	**null**	**null**
For classes	(C++)	Allowed	Not allowed

7.6 Object methods

An object's methods can be accessed using the same syntax used to access class properties:

```
Packet p = new;
status = p.current_status();
```

The above assignment to status cannot be written as follows:

```
status = current_status(p);
```

The focus in object-oriented programming is the object, in this case the packet, not the function call. Also, objects are self-contained, with their own methods for manipulating their own properties. Therefore, the object does not have to be passed as an argument to current_status(). A class's properties are freely and broadly available to the methods of the class, but each method only accesses the properties associated with its object, i.e., its instance.

7.7 Constructors

SystemVerilog does not require the complex memory allocation and deallocation of C++. Construction of an object is straightforward; and garbage collection, as in Java, is implicit and automatic. There can be no memory leaks or other subtle behavior that is so often the bane of C++ programmers.

SystemVerilog provides a mechanism for initializing an instance at the time the object is created. When an object is created, for example,

```
Packet p = new;
```

The system executes the **new** function associated with the class:

```
class Packet;
   integer command;

   function new();
      command = IDLE;
   endfunction
```

```
    endclass
```

As shown above, **new** is now being used in two very different contexts with very different semantics. The variable declaration creates an object of class `Packet`. In the course of creating this instance, the **new** function is invoked, in which any specialized initialization required can be done. The **new** function is also called the *class constructor*.

The **new** operation is defined as a function with no return type, and like any other function, it must be non-blocking. Even though **new** does not specify a return type, the left-hand side of the assignment determines the return type.

Class properties that include an initializer in their declaration are initialized before the execution of the user-defined class constructor. Thus, initializer values can be overridden by the class constructor.

Every class has a default (built-in) **new** method. The default constructor first calls its parent class constructor (**super**.**new**() as described in 7.14) and then proceeds to initialize each member of the current object to its default (or uninitialized value).

It is also possible to pass arguments to the constructor, which allows run-time customization of an object:

```
    Packet p = new(STARTUP, $random, $time);
```

where the **new** initialization task in `Packet` might now look like the following:

```
    function new(int cmd = IDLE, bit[12:0] adrs = 0, int cmd_time );
       command = cmd;
       address = adrs;
       time_requested = cmd_time;
    endfunction
```

The conventions for arguments are the same as for any other procedural subroutine calls, such as the use of default arguments.

7.8 Static class properties

The previous examples have only declared instance class properties. Each instance of the class (i.e., each object of type `Packet`) has its own copy of each of its six variables. Sometimes only one version of a variable is required to be shared by all instances. These class properties are created using the keyword **static**. Thus, for example, in following case, all instances of a class need access to a common file descriptor:

```
    class Packet ;
       static integer fileId = $fopen( "data", "r" );
```

Now, `fileID` shall be created and initialized once. Thereafter, every `Packet` object can access the file descriptor in the usual way:

```
    Packet p;
    c = $fgetc( p.fileID );
```

The static class properties can be used without creating an object of that type.

7.9 Static methods

Methods can be declared as **static**. A static method is subject to all the class scoping and access rules, but behaves like a regular subroutine that can be called outside the class, even with no class instantiation. A static method has no access to nonstatic members (class properties or methods), but it can directly access static class properties or call static methods of the same class. Access to nonstatic members or to the special **this** handle within the body of a static method is illegal and results in a compiler error. Static methods cannot be virtual.

```
class id;
    static int current = 0;
    static function int next_id();
        next_id = ++current; // OK to access static class property
    endfunction
endclass
```

A static method is different from a method with static lifetime. The former refers to the lifetime of the method within the class, while the latter refers to the lifetime of the arguments and variables within the task.

```
class TwoTasks;
    static task foo(); ... endtask   // static class method with
                                     // automatic variable lifetime

    task static bar(); ... endtask   // nonstatic class method with
                                     // static variable lifetime
endclass
```

By default, class methods have automatic lifetime for their arguments and variables.

7.10 This

The **this** keyword is used to unambiguously refer to class properties or methods of the current instance. The **this** keyword denotes a predefined object handle that refers to the object that was used to invoke the subroutine that **this** is used within. The **this** keyword shall only be used within nonstatic class methods; otherwise, an error shall be issued. For example, the following declaration is a common way to write an initialization task:

```
class Demo ;
    integer x;

    function new (integer x)
        this.x = x;
    endfunction
endclass
```

The x is now both a property of the class and an argument to the function **new**. In the function **new**, an unqualified reference to x shall be resolved by looking at the innermost scope, in this case, the subroutine argument declaration. To access the instance class property, it is qualified with the **this** keyword, to refer to the current instance.

NOTE—In writing methods, members can be qualified with **this** to refer to the current instance, but it is usually unnecessary.

7.11 Assignment, renaming, and copying

Declaring a class variable only creates the name by which the object is known. Thus,

```
Packet p1;
```

creates a variable, `p1`, that can hold the handle of an object of class `Packet`, but the initial value of `p1` is **null**. The object does not exist, and `p1` does not contain an actual handle, until an instance of type `Packet` is created:

```
p1 = new;
```

Thus, if another variable is declared and assigned the old handle, `p1`, to the new one, as in

```
Packet p2;
p2 = p1;
```

then there is still only one object, which can be referred to with either the name `p1` or `p2`. In this example, **new** was executed only once; therefore, only one object has been created.

If, however, the example above is rewritten as shown below, a copy of `p1` shall be made:

```
Packet p1;
Packet p2;
p1 = new;
p2 = new p1;
```

The last statement has **new** executing a second time, thus creating a new object p2, whose class properties are copied from p1. This is known as a *shallow copy*. All of the variables are copied across integers, strings, instance handles, etc. Objects, however, are not copied, only their handles; as before, two names for the same object have been created. This is true even if the class declaration includes the instantiation operator **new**:

```
class A ;
   integer j = 5;
endclass

class B ;
   integer i = 1;
   A a = new;
endclass

function integer test;
   B b1 = new;       // Create an object of class B
   B b2 = new b1;    // Create an object that is a copy of b1
   b2.i = 10;        // i is changed in b2, but not in b1
   b2.a.j = 50;      // change a.j, shared by both b1 and b2
   test = b1.i;      // test is set to 1 (b1.i has not changed)
   test = b1.a.j;    // test is set to 50 (a.j has changed)
endfunction
```

Several things are noteworthy. First, class properties and instantiated objects can be initialized directly in a class declaration. Second, the shallow copy does not copy objects. Third, instance qualifications can be chained as needed to reach into objects or to reach through objects:

```
b1.a.j                    // reaches into a, which is a property of b1
```

```
    p.next.next.next.val    // chain through a sequence of handles to get to val
```

To do a full (deep) copy, where everything (including nested objects) is copied, custom code is typically needed. For example:

```
    Packet p1 = new;
    Packet p2 = new;
    p2.copy(p1);
```

where `copy(Packet p)` is a custom method written to copy the object specified as its argument into its instance.

7.12 Inheritance and subclasses

The previous subclauses defined a class called `Packet`. This class can be extended so that the packets can be chained together into a list. One solution would be to create a new class called `LinkedPacket` that contains a variable of type `Packet` called `packet_c`.

To refer to a class property of `Packet`, the variable `packet_c` needs to be referenced.

```
    class LinkedPacket;
        Packet packet_c;
        LinkedPacket next;

        function LinkedPacket get_next();
            get_next = next;
        endfunction
    endclass
```

Because `LinkedPacket` is a specialization of `Packet`, a more elegant solution is to extend the class creating a new subclass that *inherits* the members of the parent class. Thus, for example:

```
    class LinkedPacket extends Packet;
        LinkedPacket next;

        function LinkedPacket get_next();
            get_next = next;
        endfunction
    endclass
```

Now, all of the methods and class properties of `Packet` are part of `LinkedPacket` (as if they were defined in `LinkedPacket`), and `LinkedPacket` has additional class properties and methods.

The parent's methods can also be overridden to change their definitions.

The mechanism provided by SystemVerilog is called *single inheritance,* that is, each class is derived from a single parent class.

7.13 Overridden members

Subclass objects are also legal representative objects of their parent classes. For example, every `Linked-Packet` object is a perfectly legal `Packet` object.

The handle of a `LinkedPacket` object can be assigned to a `Packet` variable:

```
LinkedPacket lp = new;
Packet p = lp;
```

In this case, references to p access the methods and class properties of the Packet class. So, for example, if class properties and methods in LinkedPacket are overridden, these overridden members referred to through p get the original members in the Packet class. From p, **new** and all overridden members in LinkedPacket are now hidden.

```
class Packet;
    integer i = 1;
    function integer get();
        get = i;
    endfunction
endclass

class LinkedPacket extends Packet;
    integer i = 2;
    function integer get();
        get = -i;
    endfunction
endclass

LinkedPacket lp = new;
Packet p = lp;
j = p.i;              // j = 1, not 2
j = p.get();          // j = 1, not -1 or -2
```

To call the overridden method via a parent class object (p in the example), the method needs to be declared **virtual** (see 7.19).

7.14 Super

The **super** keyword is used from within a derived class to refer to members of the parent class. It is necessary to use **super** to access members of a parent class when those members are overridden by the derived class.

```
class Packet;                           //parent class
    integer value;
    function integer delay();
        delay = value * value;
    endfunction
endclass

class LinkedPacket extends Packet;     //derived class
    integer value;
    function integer delay();
        delay = super.delay()+ value * super.value;
    endfunction
endclass
```

The member can be a member declared a level up or be inherited by the class one level up. There is no way to reach higher (for example, super.super.count is not allowed).

Subclasses (or derived classes) are classes that are extensions of the current class whereas superclasses (parent classes or base classes) are classes from which the current class is extended, beginning with the original base class.

When using the **super** within **new**, **super.new** shall be the first statement executed in the constructor. This is because the superclass must be initialized before the current class and, if the user code does not provide an initialization, the compiler shall insert a call to **super.new** automatically.

7.15 Casting

It is always legal to assign a subclass variable to a variable of a class higher in the inheritance tree. It is never legal to directly assign a superclass variable to a variable of one of its subclasses. However, it is legal to assign a superclass handle to a subclass variable if the superclass handle refers to an object of the given subclass.

To check whether the assignment is legal, the dynamic cast function $cast() is used (see 4.15).

The syntax for $cast() is as follows:

```
task $cast( singular dest_handle, singular source_handle );
```

or

```
function int $cast( singular dest_handle, singular source_handle );
```

When used with object handles, $cast() checks the hierarchy tree (super and subclasses) of the source_expr to see whether it contains the class of *dest_handle*. If it does, $cast() does the assignment. Otherwise, the error handling is as described in 4.15.

7.16 Chaining constructors

When a subclass is instantiated, the class method **new**() is invoked. The first action that **new**() takes, before any code defined in the function is evaluated, is to invoke the **new**() method of its superclass and so on up the inheritance hierarchy. Thus, all the constructors are called, in the proper order, beginning with the root base class and ending with the current class.

If the initialization method of the superclass requires arguments, there are two choices: to always supply the same arguments or to use the **super** keyword. If the arguments are always the same, then they can be specified at the time the class is extended:

```
class EtherPacket extends Packet(5);
```

This passes 5 to the **new** routine associated with Packet.

A more general approach is to use the **super** keyword, to call the superclass constructor:

```
function new();
   super.new(5);
endfunction
```

To use this approach, **super.new**(…) must be the first executable statement in the function **new**.

7.17 Data hiding and encapsulation

So far, all class properties and methods have been made available to the outside world without restriction. Often, it is desirable to restrict access to class properties and methods from outside the class by hiding their names. This keeps other programmers from relying on a specific implementation, and it also protects against accidental modifications to class properties that are internal to the class. When all data become hidden (i.e., being accessed only by public methods), testing and maintenance of the code become much easier.

In SystemVerilog, unqualified class properties and methods are public, available to anyone who has access to the object's name.

A member identified as **local** is available only to methods inside the class. Further, these local members are not visible within subclasses. Of course, nonlocal methods that access local class properties or methods can be inherited and work properly as methods of the subclass.

A **protected** class property or method has all of the characteristics of a **local** member, except that it can be inherited; it is visible to subclasses.

Within a class, a local method or class property of the same class can be referenced, even if it is in a different instance of the same class. For example:

```
class Packet;
   local integer i;
   function integer compare (Packet other);
      compare = (this.i == other.i);
   endfunction
endclass
```

A strict interpretation of encapsulation might say that other.i should not be visible inside of this packet because it is a local class property being referenced from outside its instance. Within the same class, however, these references are allowed. In this case, this.i shall be compared to other.i and the result of the logical comparison returned.

Class members can be identified as either **local** or **protected**; class properties can be further defined as **const**, and methods can be defined as **virtual**. There is no predefined ordering for specifying these modifiers; however, they can only appear once per member. It shall be an error to define members to be both **local** and **protected** or to duplicate any of the other modifiers.

7.18 Constant class properties

Class properties can be made read-only by a **const** declaration like any other SystemVerilog variable. However, because class objects are dynamic objects, class properties allow two forms of read-only variables: global constants and instance constants.

Global constant class properties include an initial value as part of their declaration. They are similar to other **const** variables in that they cannot be assigned a value anywhere other than in the declaration.

```
class Jumbo_Packet;
   const int max_size = 9 * 1024; // global constant
   byte payload [];
   function new( int size );
      payload = new[ size > max_size ? max_size : size ];
   endfunction
endclass
```

Instance constants do not include an initial value in their declaration, only the **const** qualifier. This type of constant can be assigned a value at run time, but the assignment can only be done once in the corresponding class constructor.

```
class Big_Packet;
   const int size; // instance constant
   byte payload [];
   function new();
      size = $random % 4096; //one assignment in new -> ok
      payload = new[ size ];
   endfunction
endclass
```

Typically, global constants are also declared **static** because they are the same for all instances of the class. However, an instance constant cannot be declared **static** because that would disallow all assignments in the constructor.

7.19 Abstract classes and virtual methods

A set of classes can be created that can be viewed as all being derived from a common base class. For example, a common base class of type BasePacket that sets out the structure of packets but is incomplete would never be instantiated. From this base class, however, a number of useful subclasses could be derived, such as Ethernet packets, token ring packets, GPSS packets, and satellite packets. Each of these packets might look very similar, all needing the same set of methods, but they could vary significantly in terms of their internal details.

A base class sets out the prototype for the subclasses. Because the base class is not intended to be instantiated, it can be made abstract by specifying the class to be **virtual**:

```
virtual class BasePacket;
```

Abstract classes can also have virtual methods. Virtual methods are a basic polymorphic construct. A virtual method overrides a method in all the base classes, whereas a normal method only overrides a method in that class and its descendants. One way to view this is that there is only one implementation of a virtual method per class hierarchy, and it is always the one in the latest derived class. Virtual methods provide prototypes for subroutines, i.e., all of the information generally found on the first line of a method declaration: the encapsulation criteria, the type and number of arguments, and the return type if it is needed. Later, when subclasses override virtual methods, they must follow the prototype exactly. Thus, all versions of the virtual method look identical in all subclasses:

```
virtual class BasePacket;
   virtual function integer send(bit[31:0] data);
   endfunction
endclass

class EtherPacket extends BasePacket;
   function integer send(bit[31:0] data);
      // body of the function
      ...
   endfunction
endclass
```

EtherPacket is now a class that can be instantiated. In general, if an abstract class has any virtual methods, all of the methods must be overridden (and provided with a method body) for the subclass to be instantiated. If any virtual methods have no implementation, the subclass needs to be abstract.

An abstract class can contain methods for which there is only a prototype and no implementation (i.e., an incomplete class). An abstract class cannot be instantiated; it can only be derived. Methods of normal classes can also be declared virtual. In this case, the method must have a body. If the method does have a body, then the class can be instantiated, as can its subclasses.

7.20 Polymorphism: dynamic method lookup

Polymorphism allows the use of a variable in the superclass to hold subclass objects and to reference the methods of those subclasses directly from the superclass variable. As an example, assume the base class for the `Packet` objects, `BasePacket`, defines, as virtual functions, all of the public methods that are to be generally used by its subclasses. Such methods include send, receive, and print. Even though `BasePacket` is abstract, it can still be used to declare a variable:

```
BasePacket packets[100];
```

Now, instances of various packet objects can be created and put into the array:

```
EtherPacket ep = new;   // extends BasePacket
TokenPacket tp = new;   // extends BasePacket
GPSSPacket gp = new;    // extends EtherPacket
packets[0] = ep;
packets[1] = tp;
packets[2] = gp;
```

If the data types were, for example, integers, bits, and strings, all of these types could not be stored into a single array, but with polymorphism, it can be done. In this example, because the methods were declared as **virtual**, the appropriate subclass methods can be accessed from the superclass variable, even though the compiler did not know—at compile time—what was going to be loaded into it.

For example, `packets[1]`

```
packets[1].send();
```

shall invoke the `send` method associated with the `TokenPacket` class. At run time, the system correctly binds the method from the appropriate class.

This is a typical example of polymorphism at work, providing capabilities that are far more powerful than what is found in a nonobject-oriented framework.

7.21 Class scope resolution operator ::

The class scope resolution operator `::` is used to specify an identifier defined within the scope of a class. It has the following form:

```
class_identifier :: { class_identifier :: } identifier
```

Identifiers on the left side of the class scope resolution operator `::` can be class names or package names (see 19.2).

Because classes and other scopes can have the same identifiers, the class scope resolution operator uniquely identifies a member of a particular class. In addition, to disambiguating class scope identifiers, the `::` operator also allows access to static members (class properties and methods) from outside the class, as well as access to public or protected elements of a superclass from within the derived classes.

```
class Base;
    typedef enum {bin,oct,dec,hex} radix;
    static task print( radix r, integer n ); ... endtask
endclass
...
Base b = new;
int bin = 123;
b.print( Base::bin, bin );    // Base::bin and bin are different
Base::print( Base::hex, 66 );
```

In SystemVerilog, the class scope resolution operator applies to all static elements of a class: static class properties, static methods, typedefs, enumerations, structures, unions, and nested class declarations. Class scope resolved expressions can be read (in expressions), written (in assignments or subroutines calls), or triggered off (in event expressions). They can also be used as the name of a type or a method call.

Like modules, classes are scopes and can nest. Nesting allows hiding of local names and local allocation of resources. This is often desirable when a new type is needed as part of the implementation of a class. Declaring types within a class helps prevent name collisions and the cluttering of the outer scope with symbols that are used only by that class. Type declarations nested inside a class scope are public and can be accessed outside the class.

```
class StringList;
    class Node; // Nested class for a node in a linked list.
        string name;
        Node link;
    endclass
endclass

class StringTree;
    class Node; // Nested class for a node in a binary tree.
        string name;
        Node left, right;
    endclass
endclass
// StringList::Node is different from StringTree::Node
```

The class scope resolution operator enables the following:
— Access to static public members (methods and class properties) from outside the class hierarchy.
— Access to public or protected class members of a superclass from within the derived classes.
— Access to type declarations and enumeration named constants declared inside the class from outside the class hierarchy or from within derived classes.

7.22 Out-of-block declarations

It is convenient to be able to move method definitions out of the body of the class declaration. This is done in two steps. First, within the class body, declare the method prototypes, i.e., whether it is a function or task, any qualifiers (**local**, **protected**, or **virtual**), and the full argument specification plus the **extern** qualifier. The **extern** qualifier indicates that the body of the method (its implementation) is to be found outside the declaration. Second, outside the class declaration, declare the full method (e.g., the prototype but without the qualifiers), and, to tie the method back to its class, qualify the method name with the class name and a pair of colons:

```
class Packet;
    Packet next;
    function Packet get_next();// single line
```

```
        get_next = next;
    endfunction

    // out-of-body (extern) declaration
    extern protected virtual function int send(int value);
endclass

function int Packet::send(int value);
    // dropped protected virtual, added Packet::
    // body of method
        ...
endfunction
```

The out-of-block method declaration must match the prototype declaration exactly; the only syntactical difference is that the method name is preceded by the class name and the class scope resolution operator ::.

Out-of-block declarations must be declared in the same scope as the class declaration.

7.23 Parameterized classes

It is often useful to define a generic class whose objects can be instantiated to have different array sizes or data types. This avoids writing similar code for each size or type and allows a single specification to be used for objects that are fundamentally different and (like a templated class in C++) not interchangeable.

The normal Verilog parameter mechanism is used to parameterize a class:

```
class vector #(int size = 1);
    bit [size-1:0] a;
endclass
```

Instances of this class can then be instantiated like modules or interfaces:

```
    vector #(10) vten;          // object with vector of size 10
    vector #(.size(2)) vtwo;    // object with vector of size 2
    typedef vector#(4) Vfour;   // Class with vector of size 4
```

This feature is particularly useful when using types as parameters:

```
class stack #(type T = int);
    local T items[];
    task push( T a ); ... endtask
    task pop( ref T a ); ... endtask
endclass
```

The above class defines a generic *stack* class that can be instantiated with any arbitrary type:

```
stack is;                   // default: a stack of int's
stack#(bit[1:10]) bs;       // a stack of 10-bit vector
stack#(real) rs;            // a stack of real numbers
```

Any type can be supplied as a parameter, including a user-defined type such as a `class` or `struct`.

The combination of a generic class and the actual parameter values is called a *specialization* (or *variant*). Each specialization of a class has a separate set of `static` member variables (this is consistent with C++ templated classes). To share static member variables among several class specializations, they must be placed in a nonparameterized base class.

```
    class vector #(int size = 1);
       bit [size-1:0] a;
       static int count = 0;
       function void disp_count();
          $display( "count: %d of size %d", count, size );
       endfunction
    endclass
```

The variable count in the example above can only be accessed by the corresponding disp_count method. Each specialization of the class *vector* has its own unique copy of count.

A specialization is the combination of a specific generic class with a unique set of parameters. Two sets of parameters shall be unique unless all parameters are the same as defined by the following rules:

a) A parameter is a type parameter and the two types are matching types.

b) A parameter is a value parameter and both their type and their value are the same.

All matching specializations of a particular generic class shall represent the same type. The set of matching specializations of a generic class is defined by the context of the class declaration. Because generic classes in a package are visible throughout the system, all matching specializations of a package generic class are the same type. In other contexts, such as modules or programs, each instance of the scope containing the generic class declaration creates a unique generic class, thus, defining a new set of matching specializations.

A generic class is not a type; only a concrete specialization represents a type. In the example above, the class vector becomes a concrete type only when it has had parameters applied to it, for example:

```
    typedef vector my_vector;  // use default size of 1
    vector#(6) vx;             // use size 6
```

To avoid having to repeat the specialization either in the declaration or to create parameters of that type, a **typedef** should be used:

```
    typedef vector#(4) Vfour;
    typedef stack#(Vfour) Stack4;
    Stack4 s1, s2;                // declare objects of type Stack4
```

A parameterized class can extend another parameterized class. For example:

```
    class C #(type T = bit); ... endclass          // base class
    class D1 #(type P = real) extends C;           // T is bit (the default)
    class D2 #(type P = real) extends C #(integer);     // T is integer
    class D3 #(type P = real) extends C #(P);           // T is P
```

Class D1 extends the base class C using the base class's default type (**bit**) parameter. Class D2 extends the base class C using an **integer** parameter. Class D3 extends the base class C using the parameterized type (P) with which the extended class is parameterized.

7.24 Typedef class

Sometimes a class variable needs to be declared before the class itself has been declared. For example, if two classes each need a handle to the other. When, in the course of processing the declaration for the first class, the compiler encounters the reference to the second class, that reference is undefined and the compiler flags it as an error.

This is resolved using **typedef** to provide a forward declaration for the second class:

```
typedef class C2;       // C2 is declared to be of type class
class C1;
      C2 c;
endclass
class C2;
   C1 c;
endclass
```

In this example, C2 is declared to be of type **class**, a fact that is reenforced later in the source code. In SystemVerilog, the **class** construct always creates a type and does not require a **typedef** declaration for that purpose (as in **typedef class** …). This is consistent with common C++ use.

In the preceding example, the **class** keyword in the statement **typedef class** C2; is not necessary and is used only for documentation purposes. The statement **typedef** C2; is equivalent and shall work the same way.

7.25 Classes and structures

SystemVerilog adds the object-oriented **class** construct. On the surface, it might appear that **class** and **struct** provide equivalent functionality, and only one of them is needed. However, that is not true; **class** differs from **struct** in three fundamental ways:

a) SystemVerilog structs are strictly static objects; they are created either in a static memory location (global or module scope) or on the stack of an automatic task. Conversely, SystemVerilog objects (i.e., class instances) are exclusively dynamic; their declaration does not create the object. Creating an object is done by calling **new**.

b) SystemVerilog objects are implemented using handles, thereby providing C-like pointer functionality. But, SystemVerilog disallows casting handles onto other data types; thus, unlike C, SystemVerilog handles are guaranteed to be safe.

c) SystemVerilog objects form the basis of an Object-Oriented data abstraction that provides true polymorphism. Class inheritance, abstract classes, and dynamic casting are powerful mechanisms that go way beyond the mere encapsulation mechanism provided by structs.

7.26 Memory management

Memory for objects, strings, and dynamic and associative arrays is allocated dynamically. When objects are created, SystemVerilog allocates more memory. When an object is no longer needed, SystemVerilog automatically reclaims the memory, making it available for reuse. The automatic memory management system is an integral part of SystemVerilog. Without automatic memory management, SystemVerilog's multi-threaded, reentrant environment creates many opportunities for users to run into problems. A manual memory management system, such as the one provided by C's malloc and free, would not be sufficient.

For example, consider the following example:

```
myClass obj = new;
fork
   task1( obj );
   task2( obj );
join_none
```

In this example, the main process (the one that forks off the two tasks) does not know when the two processes might be done using the object obj. Similarly, neither task1 nor task2 knows when any of the

other two processes will no longer be using the object `obj`. It is evident from this simple example that no single process has enough information to determine when it is safe to free the object. The only two options available to the user are as follows:

— Play it safe and never reclaim the object, or

— Add some form of reference count that can be used to determine when it might be safe to reclaim the object.

Adopting the first option can cause the system to quickly run out of memory. The second option places a large burden on users, who, in addition to managing their testbench, must also manage the memory using less than ideal schemes. To avoid these shortcomings, SystemVerilog manages all dynamic memory automatically. Users do not need to worry about dangling references, premature deallocation, or memory leaks. The system shall automatically reclaim any object that is no longer being used. In the example above, all that users do is assign **null** to the handle `obj` when they no longer need it. Similarly, when an object goes out of scope, the system implicitly assigns **null** to the object.

8. Operators and expressions

8.1 Introduction

NOTE—The SystemVerilog operators are a combination of Verilog and C operators. In both languages, the type and size of the operands are fixed, and hence the operator is of a fixed type and size. The fixed type and size of operators are preserved in SystemVerilog. This allows efficient code generation.

Verilog does not have assignment operators or increment and decrement operators. SystemVerilog includes the C assignment operators, such as **+=**, and the C increment and decrement operators, **++** and **--**.

Verilog-2001 added signed nets and **reg** variables and added signed based literals. There is a difference in the rules for combining signed and unsigned integers between Verilog and C. SystemVerilog uses the Verilog rules.

8.2 Operator syntax

---

assignment_operator ::=   *// from A.6.2*
     = | **+=** | **-=** | ***=** | **/=** | **%=** | **&=** | **|=** | **^=** | **<<=** | **>>=** | **<<<=** | **>>>=**

conditional_expression ::=   *// from A.8.3*
     cond_predicate **?** { attribute_instance } expression **:** expression

unary_operator ::=   *// from A.8.6*
     + | **-** | **!** | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^** | **^~**

binary_operator ::=
     + | **-** | ***** | **/** | **%** | **==** | **!=** | **===** | **!==** | **==?** | **!=?** | **&&** | **||** | ******
     | **<** | **<=** | **>** | **>=** | **&** | **|** | **^** | **^~** | **~^** | **>>** | **<<** | **>>>** | **<<<**

inc_or_dec_operator ::= **++** | **--**

unary_module_path_operator ::=
     ! | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^** | **^~**

binary_module_path_operator ::=
     == | **!=** | **&&** | **||** | **&** | **|** | **^** | **^~** | **~^**

Syntax 8-1—Operator syntax (excerpt from Annex A)

Table 8-1—Operators and data types

operator	name	data types
=	assignment operator	any
+= -= /= *=	C-like assignment operators	integral, **real**, **shortreal**
%= &= |= ^= <<= >>= <<<= >>>=	binary logical operators	integral

Table 8-1—Operators and data types *(continued)*

operator	name	data types
? :	conditional expression	any
+ -	unary arithmetic operators	integral, **real**, **shortreal**
!	unary logical operator	integral, **real**, **shortreal**
~ & ~& | ~| ^ ~^ ^~	unary logical operators	integral
+ - * / **	arithmetic binary operators	integral, **real**, **shortreal**
% & | ^ ^~ ~^ >> << >>> <<<	binary logical operators	integral
! && ||	other binary logical operators	integral, **real**, **shortreal**
< <= > >=	relational operators	integral, **real**, **shortreal**
=== !==	case equality operators	any except **real** and **shortreal**
== !=	logical equality operators	any
==? !=?	wild card equality operators	integral
++ --	increment, decrement operators	integral, **real**, **shortreal**
inside	set membership operator	singular for the left operand

Table 8-1—Operators and data types *(continued)*

operator	name	data types
{ } {{}}	concatenation, replication operators	integral
dist	distribution operator	integral

8.3 Assignment operators

In addition to the simple assignment operator, =, SystemVerilog includes the C assignment operators and special bitwise assignment operators: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, and >>>=. An assignment operator is semantically equivalent to a blocking assignment, with the exception that any left-hand index expression is only evaluated once. For example:

```
a[i]+=2; // same as a[i] = a[i] +2;
```

In SystemVerilog, an expression can include a blocking assignment, provided it does not have a timing control. These blocking assignments must be enclosed in parentheses to avoid common mistakes such as using a=b for a==b or using a|=b for a!=b.

```
if ((a=b)) b = (a+=1);

a = (b = (c = 5));
```

The semantics of such an assignment expression is that of a function that evaluates the right-hand side, casts the right-hand side to the left-hand data type, stacks it, updates the left-hand side, and returns the stacked value. The data type of the value that is returned is the data type of the left-hand side. If the left-hand side is a concatenation, then the data type of the value that is returned shall be an unsigned integral data type whose bit length is the sum of the length of its operands.

It shall be illegal to include an assignment operator in an event expression, in an expression within a procedural continuous assignment, or in an expression that is not within a procedural statement.

SystemVerilog includes the C increment and decrement assignment operators ++i, --i, i++, and i--. These do not need parentheses when used in expressions. These increment and decrement assignment operators behave as blocking assignments.

The ordering of assignment operations relative to any other operation within an expression is undefined. An implementation can warn whenever a variable is both written and read-or-written within an integral expression or in other contexts where an implementation cannot guarantee order of evaluation. For example:

```
i = 10;
j = i++ + (i = i - 1);
```

After execution, the value of j can be 18, 19, or 20 depending upon the relative ordering of the increment and the assignment statements.

8.4 Operations on logic and bit types

All Verilog operators are defined for 4-state values. In SystemVerilog, operators may be applied to 2-state values or to a mixture of 2-state and 4-state values. The result is the same as if all values were treated as

4-state values and the Verilog operators were applied. In most cases, if all operands are 2-state, the result is in the 2-state value set. The only exceptions involve operators where Verilog produces an X result for operands in the 2-state value set (e.g., division by zero).

```
int n = 8, zero = 0;
int res = 'b01xz | n;        // res gets 'b11xz coerced to int, or 'b1100
int sum = n + n;             // sum gets 16 coerced to int, or 16
int sumx = 'x + n;           // sumx gets 'x coerced to int, or 0
int div2 = n/zero + n;       // div2 gets 'x coerced to int, or 0
integer div4 = n/zero + n;   // div4 gets 'x
```

8.5 Wild equality and wild inequality

SystemVerilog introduces the wildcard comparison operators, as described in Table 8-2 and this subclause.

Table 8-2—Wild equality and wild inequality operators

Operator	Usage	Description
==?	a ==? b	a equals b, X and Z values in b act as wildcards
!=?	a !=? b	a does not equal b, X and Z values in b act as wildcards

The wild equality operator (==?) and inequality operator (!=?) treat X and Z values in a given bit position of their right operand as a wildcard. X and Z values in the left operand are not treated as wildcards. A wildcard bit matches any bit value (0, 1, Z, or X) in the corresponding bit of the left operand being compared against it. Any other bits are compared as for the logical equality and logical inequality operators.

These operators compare operands bit for bit and return a 1-bit self-determined result. If the operands to the wild-card equality/inequality are of unequal bit length, the operands are extended in the same manner as for the logical equality/inequality operators. If the relation is true, the operator yields a 1. If the relation is false, it yields a 0. If the relation is unknown, it yields X.

The different types of equality (and inequality) operators in SystemVerilog behave differently when their operands contain unknown values (X or Z). The == and != operators may result in X if any of their operands contains an X or Z. The === and !== check the 4-state explicitly; therefore, X and Z values shall either match or mismatch, never resulting in X. The ==? and !=? operators may result in X if the left operand contains an X or Z that is not being compared with a wildcard in the right operand.

8.6 Real operators

Operands of type **shortreal** have the same operation restrictions as Verilog **real** operands. The unary operators ++ and -- can have operands of type **real** and **shortreal** (the increment or decrement is by 1.0). The assignment operators **+=**, **-=**, ***=**, **/=** can also have operands of type **real** and **shortreal**.

If any operand, except before the ? in the ternary operator, is **real**, the result is **real**. Otherwise, if any operand, except before the ? in the ternary operator, is **shortreal**, the result is **shortreal**.

Real operands can also be used in the following expressions:

```
str.realval // structure or union member
realarray[intval] // array element
```

8.7 Size

The number of bits of an expression is determined by the operands and the context, following the same rules as Verilog. In SystemVerilog, casting can be used to set the size context of an intermediate value.

With Verilog, tools can issue a warning when the left- and right-hand sides of an assignment are different sizes. Using the SystemVerilog size casting, these warnings can be prevented.

8.8 Sign

The rules for determining the signedness of SystemVerilog expression types shall be the same as for Verilog. A **shortreal** converted to an integer by type coercion shall be signed.

8.9 Operator precedence and associativity

Operator precedence and associativity are listed in <u>Table 8-3</u>. The highest precedence is listed first.

Table 8-3—Operator precedence and associativity

`()  []  ::  .`	left
`+ - ! ~ & ~&	~| ^ ~^ ^~ ++ -- (unary)`	
`**`	left
`*  /  %`	left
`+  - (binary)`	left
`<<  >>  <<<  >>>`	left
`<  <=  >  >=  inside  dist`	left
`==  !=  ===  !==  ==?  !=?`	left
`& (binary)`	left
`^  ~^  ^~ (binary)`	left
`| (binary)`	left
`&&`	left
`||`	left
`?: (conditional operator)`	right
`->`	right
`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=  <<<=  >>>=  :=  :/  <=`	none
`{}  {{}}`	concatenation

8.10 Built-in methods

SystemVerilog introduces classes and the method calling syntax, in which a task or function is called using the dot notation (`.`):

```
object.task_or_function()
```

The object uniquely identifies the data on which the task or function operates. Hence, the method concept is naturally extended to built-in types in order to add functionality that traditionally was done via system tasks or functions. Unlike system tasks, built-in methods are not prefixed with a `$` because they require no special prefix to avoid collisions with user-defined identifiers. Thus, the method syntax allows extending the language without the addition of new keywords or the cluttering of the global name space with system tasks.

Built-in methods, unlike system tasks, cannot be redefined by users via PLI tasks. Thus, only functions that users should not be allowed to redefine are good candidates for built-in method calls.

In general, a built-in method is preferred over a system task when a particular functionality applies to all data types or when it applies to a specific data type. For example:

```
dynamic_array.size, associative_array.num, and string.len
```

These are all similar concepts, but they represent different things. A dynamic array has a size, an associative array contains a given number of items, and a string has a given length. Using the same system task, such as `$size`, for all of them would be less clear and intuitive.

A built-in method can only be associated with a particular data type. Therefore, if some functionality is a simple side effect (i.e., `$stop` or `$reset`) or it operates on no specific data (i.e., `$random`), then a system task must be used.

When a function or task built-in method call specifies no arguments, the empty parenthesis, `()`, following the task or function name is optional. This is also true for tasks or functions that require arguments, when all arguments have defaults specified. For a method, this rule allows simple calls to appear as properties of the object or built-in type. Similar rules are defined for functions and tasks in 12.4.5.

8.10.1 Built-in package

SystemVerilog provides a built-in package that contains system types (e.g., classes), variables, tasks, and functions. Users cannot insert additional declarations into the built-in package. The built-in package is implicitly wildcard imported into the compilation-unit scope of every compilation unit (see 19.3). Thus, declarations in the built-in package are directly available in any other scope (like system tasks and functions) unless they are redefined by user code.

built_in_data_type ::= [**std::**] data_type_identifier	*// not in Annex A*
built_in_function_call ::= [**std::**] built_in_identifier	*// not in Annex A*

The package name **std** followed by the class scope resolution operator `::` can be used to unambiguously access names in the built-in package. For example:

```
std::sys_task();          // unambiguously call the system provided sys_task
```

Unlike system tasks and functions, tasks and functions in the built-in package need not be prefixed with a `$` to avoid collisions with user-defined identifiers. This mechanism allows functional extensions to the language in a backward compatible manner, without the addition of new keywords or polluting local name spaces.

8.11 Static prefixes

Informally, the "longest static prefix" of a select is the longest part of the select for which an analysis tool has known values following elaboration. This concept is used when describing implicit sensitivity lists (see

[11.2](#)) and when describing error conditions for drivers of logic ports (see [6.7](#)). The remainder of this clause defines what constitutes the "longest static prefix" of a select.

A field select is defined as a hierarchical name where the right-hand side of the last "." hierarchy separator identifies a field of a variable whose type is a **struct** or **union** declaration. The field select prefix is defined to be the left-hand side of the final "." hierarchy separator in a field select.

An indexing select is a single indexing operation. The indexing select prefix is either an identifier or, in the case of a multidimensional select, another indexing select. Array selects, bit-selects, part-selects, and indexed part-selects are examples of indexing selects.

The definition of a static prefix is recursive and is defined as follows:
— An identifier is a static prefix.
— A hierarchical reference to an object is a static prefix.
— A package reference to net or variable is a static prefix.
— A field select is a static prefix if the field select prefix is a static prefix.
— An indexing select is a static prefix if the indexing select prefix is a static prefix and the select expression is a constant expression.

The definition of the longest static prefix is defined as follows:
— An identifier that is not the field select prefix or indexing select prefix of an expression that is a static prefix.
— A field select that is not the field select prefix or indexing select prefix of an expression that is a static prefix.
— An indexing select that is not the field select prefix or indexing select prefix of an expression that is a static prefix.

Examples:

```
localparam p = 7;
reg [7:0] m [5:1][5:1];
integer i;

m[1][i]        // longest static prefix is m[1]

m[p][1]        // longest static prefix is m[p][1]

m[i][1]        // longest static prefix is m
```

8.12 Concatenation

Braces ({ }) are used to show concatenation, as in Verilog. The concatenation is treated as a packed vector of bits. It can be used on the left-hand side of an assignment or in an expression.

```
logic log1, log2, log3;
{log1, log2, log3} = 3'b111;
{log1, log2, log3} = {1'b1, 1'b1, 1'b1}; // same effect as 3'b111
```

Software tools can generate a warning if the concatenation width on one side of an assignment is different from the expression on the other side. The following examples can give warning of size mismatch:

```
bit [1:0] packedbits = {32'b1,32'b1}; // right-hand side is 64 bits
int i = {1'b1, 1'b1}; // right-hand side is 2 bits
```

Refer to 3.7 and 3.8 for information on initializing arrays and structures.

SystemVerilog enhances the concatenation operation to allow concatenation of data objects of type **string**. In general, if any of the operands is of the data type **string**, the concatenation is treated as a string, and all other arguments are implicitly converted to the **string** data type (as described in 4.7). String concatenation is not allowed on the left-hand side of an assignment, only as an expression.

```
string hello = "hello";
string s;
s = { hello, " ", "world" };
$display( "%s\n", s );        // displays 'hello world'
s = { s, " and goodbye" };
$display( "%s\n", s );        // displays 'hello world and goodbye'
```

The replication operator (also called a *multiple concatenation*) form of braces can also be used with data objects of type **string**. In the case of string replication, a nonconstant multiplier is allowed.

```
int n = 3;
string s = {n { "boo " }};
$display( "%s\n", s );  // displays 'boo boo boo '
```

Unlike bit concatenation, the result of a string concatenation or replication is not truncated. Instead, the destination variable (of type **string**) is resized to accommodate the resulting string.

8.13 Assignment patterns

In Verilog the assignment is the basic mechanism for placing values into data objects. SystemVerilog extends the syntax of assignment to describe patterns of assignments to structure fields and array elements.

An assignment pattern specifies a correspondence between a collection of expressions and the fields and elements in a data object or data value. An assignment pattern has no self-determined data type, but can be used as one of the sides in an assignment-like context (see below) when the other side has a self-determined data type. An assignment pattern is built from braces, keys, and expressions and is prefixed with an apostrophe. For example:

```
var int A[N] = '{default:1};
var integer i = '{31:1, 23:1, 15:1, 8:1, default:0};

typedef struct {real r, th;} C;
var C x = '{th:PI/2.0, r:1.0};
var real y, z;
```

A positional notation without keys can also be used. For example:

```
var int B[4] = '{a, b, c, d};
var C y = '{1.0, PI/2.0};
'{a, b, c, d} = B;
```

When an assignment pattern is used as the left-hand side of an assignment-like context, the positional notation shall be required; and each member expression shall have a bit-stream data type that is assignment compatible with and has the same number of bits as the data type of the corresponding element on the right-hand side.

```
assignment_pattern ::=                                                    // from A.6.7.1
        '{ expression { , expression } }
     | '{ structure_pattern_key : expression { , structure_pattern_key : expression } }
     | '{ array_pattern_key : expression { , array_pattern_key : expression } }
     | '{ constant_expression { expression { , expression } } }
structure_pattern_key ::= member_identifier | assignment_pattern_key
array_pattern_key ::= constant_expression | assignment_pattern_key
assignment_pattern_key ::= simple_type | default
assignment_pattern_expression ::=
        [ assignment_pattern_expression_type ] assignment_pattern
assignment_pattern_expression_type ::= ps_type_identifier | ps_parameter_identifier | integer_atom_type
constant_assignment_pattern_expression^35 ::= assignment_pattern_expression

35). In a constant_assignment_pattern_expression, all member expressions shall be constant expressions.
```

Syntax 8-2—Assignment patterns

An assignment-like context is as follows:

— A continuous or procedural assignment

— For a parameter with an explicit type declaration:

— A parameter value assignment in a module, interface, program, or class

— A parameter value override in the instantiation of a module, interface, or program

— A parameter value override in the instantiation of a class or in the left-hand side of a class scope operator

— A port connection to an input or output port of a module, interface, or program

— The passing of a value to a subroutine input, output, or inout port

— A return statement in a function

— A tagged union expression

— For an expression that is used as the right-hand value in an assignment-like context:

— If a parenthesized expression, then the expression within the parentheses

— If a mintypmax expression, then the colon-separated expressions

— If a conditional operator expression, then the second and third operand

— A nondefault correspondence between an expression in an assignment pattern and a field or element in a data object or data value

No other contexts shall be considered assignment-like contexts. In particular, none of the following shall be considered assignment-like contexts:

— A static cast

— A default correspondence between an expression in an assignment pattern and a field or element in a data object or data value

— A port expression in a module, interface, or program declaration

— The passing of a value to a subroutine **ref** port

— A port connection to an **inout** or **ref** port of a module, interface, or program

An assignment pattern can be used to construct or deconstruct a structure or array by prefixing the pattern with the name of a data type to form an assignment pattern expression. Unlike an assignment pattern, an assignment pattern expression has a self-determined data type and is not restricted to being one of the sides in an assignment-like context. When an assignment pattern expression is used in a right-hand expression, it shall yield the value that a variable of the data type would hold if it were initialized using the assignment pattern.

```
typedef logic [1:0] [3:0] T;
shortint'({T'{1,2}, T'{3,4}})     // yields 16'sh1234
```

When an assignment pattern expression is used in a left-hand expression, the positional notation shall be required; and each member expression shall have a bit-stream data type that is assignment compatible with and has the same number of bits as the corresponding element in the data type of the assignment pattern expression. If the right-hand expression has a self-determined data type, then it shall be assignment compatible with and have the same number of bits as the data type of the assignment pattern expression.

```
typedef byte U[3];
var U A = '{1, 2, 3};
var byte a, b, c;
U'{a, b, c} = A;
U'{c, a, b} = '{a+1, b+1, c+1};
```

An assignment pattern expression shall not be used in a port expression in a module, interface, or program declaration.

8.13.1 Array assignment patterns

Verilog uses concatenation braces to construct and deconstruct simple bit vectors. SystemVerilog adds a similar syntax to support the construction and deconstruction of arrays. Unlike in C, the expressions must match element for element, and the braces must match the array dimensions. Each expression item shall be evaluated in the context of an assignment to the type of the corresponding element in the array. In other words, the following examples do not give size warnings, unlike the similar assignments above:

```
bit unpackedbits [1:0] = '{1,1};  // no size warning as bit can be set to 1
int unpackedints [1:0] = '{1'b1, 1'b1};  // no size warning as int can be
                                         // set to 1'b1
```

A syntax resembling multiple concatenations can be used in array assignment patterns as well. Each replication represents a single dimension.

```
unpackedbits = '{2 {y}} ;         // same as '{y, y}
int n[1:2][1:3] = '{2{'{3{y}}}}; // same as '{'{y,y,y},'{y,y,y}}
```

SystemVerilog determines the context of the braces when used in the context of an assignment.

It can sometimes be useful to set array elements to a value without having to keep track of how many members there are. This can be done with the **default** keyword:

```
initial unpackedints = '{default:2};   // sets elements to 2
```

For more arrays of structures, it is useful to specify one or more matching type keys.

```
struct {int a; time b;} abkey[1:0];
abkey = '{'{a:1, b:2ns}, '{int:5, time:$time}};
```

The matching rules are as follows:

— An `index:value` specifies an explicit value for a keyed element index. The value is evaluated in the context of an assignment to the indexed element and shall be castable to its type. It shall be an error to specify the same index more than once in a single array expression.

— For `type:value`, if the element or subarray type of the array matches this type, then each element or subarray shall be set to the value. The value must be castable to the array element or subarray type. Otherwise, if the array is multidimensional, then there is a recursive descent into each subarray of the array using the rules in this clause and the type and default keys. Otherwise, if the array is an array of structures, there is a recursive descent into each element of the array using the rules for structure assignment patterns and the type and default keys. If more than one type matches the same element, the last value shall be used.

— The `default:value` applies to elements or subarrays that are not matched by either index or type key. If the type of the element or subarray is a simple bit vector type, matches the self-determined type of the value, or is not an array or structure type, then the value is evaluated in the context of each assignment to an element or subarray by the default and must be castable to the type of the element or subarray; otherwise, an error is generated. For unmatched subarrays, the type and default specifiers are applied recursively according to the rules in this clause to each of its elements or subarrays. For unmatched structure elements, the type and default keys are applied to the element according to the rules for structures.

Every element shall be covered by one of these rules.

If the type key, default key, or replication operator is used on an expression with side effects, the number of times that expression evaluates is undefined.

8.13.2 Structure assignment patterns

A structure can be constructed and deconstructed with a structure assignment pattern built from member expressions using braces and commas, with the members in declaration order. Replicate operators can be used to set the values for the exact number of members. Each member expression shall be evaluated in the context of an assignment to the type of the corresponding member in the structure. It can also be built with the names of the members

```
module mod1;

    typedef struct {
        int x;
        int y;
    } st;

    st s1;
    int k = 1;

    initial begin
        #1 s1 = '{1, 2+k};          // by position
        #1 $display( s1.x, s1.y);
        #1 s1 = '{x:2, y:3+k);      // by name
        #1 $display( s1);
        #1 $finish;
    end
endmodule
```

It can sometimes be useful to set structure members to a value without having to keep track of how many members there are or what the names are. This can be done with the **default** keyword:

```
    initial s1 = '{default:2}; // sets x and y to 2
```

The '{member:value} or '{data_type: default_value} syntax can also be used:

```
    ab abkey[1:0] = '{'{a:1, b:1.0}, '{int:2, shortreal:2.0}};
```

Use of the **default** keyword applies to members in nested structures or elements in unpacked arrays in structures.

```
    struct {
        int A;
        struct {
            int B, C;
        } BC1, BC2;
    } ABC, DEF;

    ABC = '{A:1, BC1:'{B:2, C:3}, BC2:'{B:4,C:5}};
    DEF = '{default:10};
```

To deal with the problem of members of different types, a type can be used as the key. This overrides the default for members of that type:

```
    typedef struct {
        logic [7:0] a;
        bit b;
        bit signed [31:0] c;
        string s;
    } sa;

    sa s2;
    initial s2 = '{int:1, default:0, string:""};        // set all to 0 except the
                                                        // array of bits to 1 and
                                                        // string to ""
```

Similarly, an individual member can be set to override the general default and the type default:

```
    initial #10 s1 = '{default:'1, s : ""}; // set all to 1 except s to ""
```

SystemVerilog determines the context of the braces when used in the context of an assignment.

The matching rules are as follows:
— A member:value: specifies an explicit value for a named member of the structure. The named member must be at the top level of the structure; a member with the same name in some level of sub-structure shall not be set. The value must be castable to the member type and is evaluated in the context of an assignment to the named member; otherwise, an error is generated.

— The type:value specifies an explicit value for each field in the structure whose type matches the type (see 6.9.1) and has not been set by a field name key above. If the same type key is mentioned more than once, the last value is used. The value is evaluated in the context of an assignment to the matching type.

— The default:value applies to members that are not matched by either member name or type key. If the member type is a simple bit vector type, matches the self-determined type of the value, or is not an array or structure type, then the value is evaluated in the context of each assignment to a member by the default and must be castable to the member type; otherwise, an error is generated. For unmatched structure members, the type and default specifiers are applied recursively according

to the rules in this clause to each member of the substructure. For unmatched array members, the type and default keys are applied to the array according to the rules for arrays.

Every member must be covered by one of these rules.

If the type key, default key, or replication operator is used on an expression with side effects, the number of times that expression evaluates is undefined.

8.14 Tagged union expressions and member access

```
expression ::=                                                     // from A.8.3
     ...
    | tagged_union_expression
tagged_union_expression ::=
     tagged member_identifier [ expression ]
```

Syntax 8-3—Tagged union syntax (excerpt from Annex A)

A tagged union expression (packed or unpacked) is expressed using the keyword **tagged** followed by a tagged union member identifier, followed by an expression representing the corresponding member value. For **void** members the member value expression is omitted.

Example:

```
typedef union tagged {
   void Invalid;
   int Valid;
} VInt;

VInt vi1, vi2;

vi1 = tagged Valid (23+34);   // Create Valid int
vi2 = tagged Invalid;         // Create an Invalid value
```

In the tagged union expressions below, the expressions in braces are structure assignment patterns (see 8.13.2).

```
typedef union tagged {
   struct {
      bit [4:0] reg1, reg2, regd;
   } Add;
   union tagged {
      bit [9:0] JmpU;
      struct {
         bit [1:0] cc;
         bit [9:0] addr;
      } JmpC;
   } Jmp;
} Instr;

Instr i1, i2;

// Create an Add instruction with its 3 register fields
i1 = ( e
```

```
        ? tagged Add '{ e1, 4, ed };                  // struct members by position
        : tagged Add '{ reg2:e2, regd:3, reg1:19 }); // by name (order irrelevant)

    // Create a Jump instruction, with "unconditional" sub-opcode
    i1 = tagged Jmp (tagged JmpU 239);

    // Create a Jump instruction, with "conditional" sub-opcode
    i2 = tagged Jmp (tagged JmpC '{ 2, 83 });        // inner struct by
    position
    i2 = tagged Jmp (tagged JmpC '{ cc:2, addr:83 }); // by name
```

The type of a tagged union expression must be known from its context (e.g., it is used in the right-hand side of an assignment to a variable whose type is known, or it is has a cast, or it is used inside another expression from which its type is known). The expression evaluates to a tagged union value of that type. The tagged union expression can be completely type-checked statically: the only member names allowed after the **tagged** keyword are the member names for the expression type, and the member expression must have the corresponding member type.

An uninitialized variable of tagged union type shall be undefined. This includes the tag bits. A variable of tagged union type can be initialized with a tagged union expression provided the member value expression is a legal initializer for the member type.

Members of tagged unions can be read or assigned using the usual dot notation. Such accesses are completely type-checked, i.e., the value read or assigned must be consistent with the current tag. In general, this can require a run-time check. An attempt to read or assign a value whose type is inconsistent with the tag results in a run-time error.

All the following examples are legal only if the instruction variable i1 currently has tag Add:

```
    x = i1.Add.reg1;
    i1.Add = '{19, 4, 3};
    i1.Add.reg2 = 4;
```

8.15 Aggregate expressions

Unpacked structure and array data objects, as well as unpacked structure and array constructors, can all be used as aggregate expressions. A multi-element slice of an unpacked array can also be used as an aggregate expression.

Aggregate expressions can be copied in an assignment, through a port, or as an argument to a task or function. Aggregate expressions can also be compared with equality or inequality operators. To be copied or compared, the type of an aggregate expression must be equivalent. See 6.9.2.

8.16 Operator overloading

There are various kinds of arithmetic that can be useful: saturating, arbitrary size floating point, carry save, etc. It is convenient to use the normal arithmetic operators for readability, rather than relying on function calls.

overload_declaration ::=	// from A.2.8

```
overload_declaration ::=                                                    // from A.2.8
        bind overload_operator function data_type function_identifier ( overload_proto_formals ) ;
overload_operator ::= + | ++ | – | – – | * | ** | / | % | == | != | < | <= | > | >= | =
overload_proto_formals ::= data_type {, data_type}
```

Syntax 8-4—Operator overloading syntax (excerpt from Annex A)

The overload declaration allows the arithmetic operators to be applied to data types that are normally illegal for them, such as unpacked structures. It does not change the meaning of the operators for data types where it is legal to apply them. In other words, such code does not change behavior when operator overloading is used.

The overload declaration links an operator to a function prototype. The arguments are matched, and the data type of the result is then checked. Multiple functions can have the same arguments and different return data types. If no expected data type exists because the operator is in a self-determined context, then a cast must be used to select the correct function. Similarly if more than one expected data type is possible, due to nested operators, and could match more than one function, a cast must be used to select the correct function.

An expected result data type exists in any of the following contexts:
— Right-hand side of an assignment or assignment expression
— Actual input argument of a task or function call
— Input port connection of a module, interface, or program
— Actual parameter to a module, interface, program, or class
— Relational operator with unambiguous comparison
— Inside a cast

For example, suppose there is a structure type float:

```
typedef struct {
   bit sign;
   bit [3:0] exponent;
   bit [10:0] mantissa;
} float;
```

The + operator can be applied to this structure by invoking a function as indicated in the overloading declarations below:

```
bind + function float faddif(int, float);
bind + function float faddfi(float, int);
bind + function float faddrf(real, float);
bind + function float faddrf(shortreal, float);
bind + function float faddfr(float, real);
bind + function float faddfr(float, shortreal);
bind + function float faddff(float, float);
bind + function float fcopyf(float); // unary +
bind + function float fcopyi(int); // unary +
bind + function float fcopyr(real); // unary +
bind + function float fcopyr(shortreal); // unary +

float A, B, C, D;
assign A = B + C;    //equivalent to A = faddff(B, C);
assign D = A + 1.0;  //equivalent to A = faddfr(A, 1.0);
```

The overloading declaration links the + operator to each function prototype according to the equivalent argument data types in the overloaded expression, which normally must match exactly. The exception is if the actual argument is an integral type and there is only one prototype with a corresponding integral argument, the actual is implicitly cast to the data type in the prototype.

A function prototype does not need to match the actual function declaration exactly. If it does not, then the normal implicit casting rules apply when calling the function. For example, the `fcopyi` function can be defined with an **int** argument:

```
function float fcopyi (int i);
    float o;
    o.sign = i[31];
    o.exponent = 0;
    o.mantissa = 0;
    …
    return o;
endfunction
```

Overloading the assignment operator also serves to overload implicit assignments or casting. Here these are using the same functions as the unary +.

```
bind = function float fcopyi(int);          // cast int to float
bind = function float fcopyr(real);         // cast real to float
bind = function float fcopyr(shortreal);    // cast shortreal to float
```

The operators that can be overloaded are the arithmetic operators, the relational operators, and assignment. The assignment operator from a float to a float cannot be overloaded above because it is already legal in the three preceding **bind** statements. Similarly, equality and inequality between floats cannot be overloaded.

No format can be assumed for 0 or 1; therefore, the user cannot rely on subtraction to give equality or on addition to give increment. Similarly, no format can be assumed for positive or negative; therefore, comparison must be explicitly coded.

An assignment operator such as += is automatically built from both the + and = operators successively, where the = has its normal meaning. For example:

```
float A, B;
bind + function float faddff(float, float);
always @(posedge clock) A += B;     // equivalent to A = A + B
```

The scope and visibility of the overload declaration follows the same search rules as a data declaration. The overload declaration must be defined before use in a scope that is visible. The function bound by the overload declaration uses the same scope search rules as a function enable from the scope where the operator is invoked.

8.17 Streaming operators (pack/unpack)

The bit-stream casting described in 4.16 is most useful when the conversion operation can be easily expressed using only a type cast and the specific ordering of the bit stream is not important. Sometimes, however, a stream that matches a particular machine organization is required. The streaming operators perform packing of bit-stream types (see 4.16) into a sequence of bits in a user-specified order. When used in the left-hand side, the streaming operators perform the reverse operation, i.e., unpack a stream of bits into one or more variables. If the data being packed contains any 4-state types, the result of a pack operation is a 4-state stream; otherwise, the result of a pack is a 2-state stream. Unpacking a 4-state stream into a 2-state type is done by a cast to a 2-state variable, and vice versa.

The syntax of the bit-stream concatenation is as follows:

```
streaming_concatenation ::= { stream_operator [ slice_size ] stream_concatenation }          // from A.8.
stream_operator ::= >> | <<
slice_size ::= simple_type | constant_expression
stream_concatenation ::= { stream_expression { , stream_expression } }
stream_expression ::= expression [ with [ array_range_expression ] ]
array_range_expression ::=
        expression
      | expression : expression
      | expression +: expression
      | expression -: expression
primary ::=
        ...
      | streaming_concatenation
```

Syntax 8-5—Streaming concatenation syntax (excerpt from Annex A)

The stream operator determines the order in which data are streamed: >> causes data to be streamed in left-to-right order, while << causes data to be streamed in right-to-left order. If a slice size is specified, then the data to be streamed are first broken up into slices with the specified number of bits, and then the slices are streamed in the specified order. If a slice size is not specified, the default is 1 (or bit). If, as a result of slicing, the last slice is less than the slice width, then no padding is added.

For example:

```
int j = { "A", "B", "C", "D" };
{ >> {j}}                    // generates stream "A" "B" "C" "D"
{ << byte {j}}               // generates stream "D" "C" "B" "A" (little endian)
{ << 16 {j}}                 // generates stream "C" "D" "A" "B"
{ << { 8'b0011_0101 }}       // generates stream 'b1010_1100 (bit reverse)
{ << 4 { 6'b11_0101 }}       // generates stream 'b0101_11
{ >> 4 { 6'b11_0101 }}       // generates stream 'b1101_01 (same)
{ << 2 { { << { 4'b1101 }} }}  // generates stream 'b1110
```

The streaming operators operate directly on integral types and streams. When applied to unpacked aggregate types, such as unpacked arrays, unpacked structures, or classes, they recursively traverse the data in depth-first order until reaching an integral type. A multidimensional packed array is thus treated as a single integral type, whereas an unpacked array of packed items causes each packed item to be streamed individually. The streaming operators can only process bit-stream types; any other types shall generate an error.

The result of the pack operation can be assigned directly to any bit-stream type variable. If the left-hand side represents a fixed-size variable and the stream is larger than the variable, an error will be generated. If the variable is larger than the stream, the stream is left-justified and zero-filled on the right. If the left-hand side represents a dynamically sized variable, such as a queue or dynamic array, the variable is resized to accommodate the entire stream. If after resizing, the variable is larger than the stream, the stream is left-justified and zero-filled on the right. The stream is not an integral value; to participate in an expression, a cast is required.

The unpack operation accepts any bit-stream type on the right-hand side, including a stream. The right-hand data being unpacked are allowed to have more bits than are consumed by the unpack operation. However, if more bits are needed than are provided by the right-hand expression, an error is generated.

For example:

```
int a, b, c;
logic [10:0] up [3:0];
logic [11:1] p1, p2, p3, p4;
bit [96:1] y = {>>{ a, b, c }};      // OK: pack a, b, c
int j = {>>{ a, b, c }};             // error: j is 32 bits < 96 bits
bit [99:0] d = {>>{ a, b, c }};      // OK: b is padded with 4 bits
{>>{ a, b, c }} = 23'b1;             // error: too few bits in stream
{>>{ a, b, c }} = 96'b1;             // OK: unpack a = 0, b = 0, c = 1
{>>{ a, b, c }} = 100'b1;            // OK: unpack as above (4 bits unread)
{ >> {p1, p2, p3, p4}} = up;         // OK: unpack p1 = up[3], p2 = up[2],
                                     // p3 = up[1], p4 = up[0]
```

When applied to a class handle (i.e., an object), the streaming operator shall stream the contents of the object, and not the handle itself. Class items are streamed in declaration order; extended class items shall follow the items of their superclass. Embedded class handles are streamed as other aggregate types: they are recursively traversed in depth-first order until reaching integral types. A null class handle shall be ignored (not streamed), and a warning may be issued. Null handles are skipped by both the pack and unpack operators; therefore, the unpack operation shall not create class objects. If a particular object hierarchy is to be reconstructed from a stream, the object hierarchy into which the stream is unpacked must be created before the streaming operator is applied. The result of streaming an object hierarchy that contains cycles shall be undefined, and an error may be issued.

8.17.1 Streaming dynamically sized data

If the unpack operation includes unbounded dynamically sized types, the process is greedy (as in a cast): the first dynamically sized item is resized to accept all the available data (excluding subsequent fixed-size items) in the stream; any remaining dynamically sized items are left empty. This mechanism is sufficient to unpack a packet-sized stream that contains only one dynamically sized data item. However, when the stream contains multiple variable-size data packets, or each data packet contains more than one variable-size data item, or the size of the data to be unpacked is stored in the middle of the stream, this mechanism can become cumbersome and error-prone. To overcome these problems, the unpack operation allows a **with** expression to explicitly specify the extent of a variable-size field within the unpack operation.

The syntax of the **with** expression is as follows:

```
stream_expression ::= expression [ with [ array_range_expression ] ]        // from A.8.1

array_range_expression ::=
      expression
    | expression : expression
    | expression +: expression
    | expression -: expression
```

Syntax 8-6—With expression syntax (excerpt from Annex A)

The array range expression within the **with** construct must be of integral type and evaluate to values that lie within the bounds of a fixed-size array or to positive values for dynamic arrays or queues. The expression before the **with** can be any one-dimensional unpacked array (including a queue). The expression within the **with** is evaluated immediately before its corresponding array is streamed (i.e., packed or unpacked). Thus, the expression can refer to data that are unpacked by the same operator but before the array. If the expression refers to variables that are unpacked after the corresponding array (to the right of the array), then the expression is evaluated using the previous values of the variables.

When used within the context of an unpack operation and the array is a variable-size array, it shall be resized to accommodate the range expression. If the array is a fixed-size array and the range expression evaluates to a range outside the extent of the array, only the range that lies within the array is unpacked and an error is generated. If the range expression evaluates to a range smaller than the extent of the array (fixed or variable size), only the specified items are unpacked into the designated array locations; the remainder of the array is unmodified.

When used within the context of a pack (on the right-hand side), it behaves the same as an array slice. The specified number of array items are packed into the stream. If the range expression evaluates to a range smaller than the extent of the array, only the specified array items are streamed. If the range expression evaluates to a range greater than the extent of the array size, the entire array is streamed, and the remaining items are generated using the default value (as described in Table 5-1) for the given array.

For example, the code below uses streaming operators to model a packet transfer over a byte stream that uses little-endian encoding:

```
byte stream[$];   // byte stream

class Packet
   rand int header;
   rand int len;
   rand byte payload[];
   int crc;

   constraint G { len > 1; payload.size == len ; }

   function void post_randomize; crc = payload.sum; endfunction
endclass

...
send: begin                    // Create random packet and transmit
   byte q[$];
   Packet p = new;
   void'(p.randomize());
   q = {<< byte{p.header, p.len, p.payload, p.crc}}; // pack
   stream = {stream, q};                             // append to stream
end

...
receive: begin       // Receive packet, unpack, and remove
   byte q[$];
   Packet p = new;
   {<< byte{ p.header, p.len, p.payload with [0 +: p.len], p.crc }} = stream;
   stream = stream[ $bits(p) / 8 : $ ];   // remove packet
end
```

In the example above, the pack operation could have been written as either:

```
q = {<<byte{p.header, p.len, p.payload with [0 +: p.len], p.crc}};
```

or

```
q = {<<byte{p.header, p.len, p.payload with [0 : p.len-1], p.crc}};
```

or

```
q = {<<byte{p}};
```

The result in this case would be the same because `p.len` is the size of `p.payload` as specified by the constraint.

8.18 Conditional operator

conditional_expression ::=   // from A.8.3
      cond_predicate **?** { attribute_instance } expression **:** expression

cond_predicate ::=   // from A.6.6
      expression_or_cond_pattern { **&&&** expression_or_cond_pattern }

expression_or_cond_pattern ::=
      expression | cond_pattern

cond_pattern ::= expression **matches** pattern

Syntax 8-7—Conditional operator syntax (excerpt from Annex A)

This subclause describes the traditional notation where *cond_predicate* is just a single expression. SystemVerilog also allows *cond_predicate* to perform pattern matching, and this is described in 10.4.

As defined in Verilog, if *cond_predicate* is true, the operator returns first *expression*; if false, it returns second *expression*. If *cond_predicate* evaluates to an ambiguous value (x or z), then both first *expression* and second *expression* shall be evaluated, and their results shall be combined bit by bit.

SystemVerilog extends the conditional operator to nonintegral types and aggregate expressions using the following rules:

— If both first *expression* and second *expression* are of integral type, the operation proceeds as defined.
— If first *expression* or second *expression* is an integral type and the opposing expression can be implicitly cast to an integral type, the cast is made and proceeds as defined.
— For all other cases, the type of first *expression* and second *expression* must be equivalent.

If *cond_predicate* evaluates to an ambiguous value, then both first *expression* and second *expression* shall be evaluated, and their results shall be combined element by element. If the elements match, the element is returned. If they do not match, then the default-uninitialized value for that element's type shall be returned.

8.19 Set membership

SystemVerilog supports singular value sets and set membership operators.

The syntax for the set membership operator is as follows:

inside_expression ::= expression **inside {** open_range_list **}**   *// from A.8.3*

Syntax 8-8—Inside expression syntax (excerpt from Annex A)

The *expression* on the left-hand side of the **inside** operator is any singular expression.

The set-membership open_range_list on the right-hand side of the inside operator is a comma-separated list of expressions or ranges. If an expression in the list is an aggregate array, its elements are traversed by descending into the array until reaching a singular value. The members of the set are scanned until a match is found and the operation returns `1'b1`. Values can be repeated; therefore, values and value ranges can overlap. The order of evaluation of the expressions and ranges is nondeterministic.

```
int a, b, c;
if ( a inside {b, c} ) ...
int array [$] = '{3,4,5};
if ( ex inside {1, 2, array} ) ... // same as { 1, 2, 3, 4, 5}
```

The **inside** operator uses the equality (`==`) operator on nonintegral expressions to perform the comparison. If no match is found, the **inside** operator returns `1'b0`. Integral expressions use the wildcard equality (`==?`) operator so that an `x` or `z` bit in a value in the set is treated as a do-not-care in that bit position (see 8.5). As with wildcard equality, an `x` or `z` in the expression on the left-hand side of the inside operator is not treated as a do-not-care.

```
logic [2:0] val;
while ( val inside {3'b1?1} ) ... // matches 3'b101, 3'b111, 3'b1x1, 3'b1z1
```

If no match is found, but some of the comparisons result in x, the inside operator shall return `1'bx`. The return value is effectively the or reduction of all the comparisons in the set with the expression on the left-hand side.

```
wire r;
assign r=3'bz11 inside {3'b1?1, 3'b011};  // r = 1'bx
```

A range can be specified with a low and high bound enclosed by square braces `[ ]` and separated by a colon (`:`), as in `[low_bound:high_bound]`. A bound specified by `$` shall represent the lowest or highest value for the type of the expression on the left-hand side. A match is found if the expression on the left-hand side is inclusively within the range. When specifying a range, the expressions must be of a singular type for which the relation operators (`<=, >=`) are defined. If the bound to the left of the colon is greater than the bound to the right, the range is empty and contains no values.

For example:

```
bit ba = a inside { [16:23], [32:47] };
string I;
if (I inside {["a rock":"hard place"]}) ...
    // I between "a rock" and a "hard place"
```

9. Scheduling semantics

9.1 Execution of a hardware model and its verification environment

NOTE—This clause gives an overview of the interactions between these elements, especially with respect to the scheduling and execution of events. Although SystemVerilog is not limited to simulation, the semantics of the language is defined for event-directed simulation, and other uses of the HDL are abstracted from this base definition.

9.2 Event simulation

The SystemVerilog language is defined in terms of a discrete event execution model. The discrete event simulation is described in more detail in this clause to provide a context to describe the meaning and valid interpretation of SystemVerilog constructs. These resulting definitions provide the standard SystemVerilog reference algorithm for simulation, which all compliant simulators shall implement. Within the following event execution model definitions, there is a great deal of choice, and differences in some details of execution are to be expected between different simulators. In addition, SystemVerilog simulators are free to use different algorithms from those described in this clause, provided the user-visible effect is consistent with the reference algorithm.

A SystemVerilog description consists of connected threads of execution or processes. Processes are objects that can be evaluated, that can have state, and that can respond to changes on their inputs to produce outputs. Processes are concurrently scheduled elements, such as `initial` blocks. Example of processes include, but are not limited to, primitives; `initial`, `always`, `always_comb`, `always_latch`, and `always_ff` procedural blocks; continuous assignments; asynchronous tasks; and procedural assignment statements.

Every change in state of a net or variable in the system description being simulated is considered an update event.

Processes are sensitive to update events. When an update event is executed, all the processes that are sensitive to that event are considered for evaluation in an arbitrary order. The evaluation of a process is also an event, known as an *evaluation event*.

Evaluation events also include PLI callbacks, which are points in the execution model where PLI application routines can be called from the simulation kernel.

In addition to events, another key aspect of a simulator is time. The term *simulation time* is used to refer to the time value maintained by the simulator to model the actual time it would take for the system description being simulated. The term *time* is used interchangeably with simulation time in this clause.

To fully support clear and predictable interactions, a single time slot is divided into multiple regions where events can be scheduled that provide for an ordering of particular types of execution. This allows properties and checkers to sample data when the design under test is in a stable state. Property expressions can be safely evaluated, and testbenches can react to both properties and checkers with zero delay, all in a predictable manner. This same mechanism also allows for nonzero delays in the design, clock propagation, and/or stimulus and response code to be mixed freely and consistently with cycle-accurate descriptions.

9.3 The stratified event scheduler

A compliant SystemVerilog simulator must maintain some form of data structure that allows events to be dynamically scheduled, executed, and removed as the simulator advances through time. The data structure is normally implemented as a time-ordered set of linked lists, which are divided and subdivided in a well-defined manner.

The first division is by time. Every event has one and only one simulation execution time, which at any given point during simulation can be the current time or some future time. All scheduled events at a specific time define a time slot. Simulation proceeds by executing and removing all events in the current simulation time slot before moving on to the next nonempty time slot, in time order. This procedure guarantees that the simulator never goes backwards in time.

A time slot is divided into a set of ordered regions:

    a)   Preponed

    b)   Pre-active

    c)   Active

    d)   Inactive

    e)   Pre-NBA

    f)   NBA

    g)   Post-NBA

    h)   Observed

    i)   Post-observed

    j)   Reactive

    k)   Re-inactive

    l)   Pre-postponed

    m)   Postponed

The purpose of dividing a time slot into these ordered regions is to provide predictable interactions between the design and testbench code.

Except for the Observed, Reactive, and Re-inactive regions and the Post-observed PLI region, these regions essentially encompass IEEE 1364 reference model for simulation, with exactly the same level of determinism. In other words, legacy Verilog code shall continue to run correctly without modification within the new mechanism. The Postponed region is where the monitoring of signals, and other similar events, takes place. No new value changes are allowed to happen in the time slot once the Postponed region is reached.

The Observed, Reactive, and Re-inactive regions are new in this standard, and events are only scheduled into these new regions from new language constructs.

The Observed region is for the evaluation of the property expressions when they are triggered. A criterion for this determinism is that the property evaluations must only occur once in any clock triggering time slot. During the property evaluation, pass/fail code shall be scheduled in the Reactive region of the current time slot. PLI callbacks are not allowed in the Observed region.

The new #1**step** sampling delay provides the ability to sample data immediately before entering the current time slot and is a preferred construct over other equivalent constructs because it allows the 1**step** time delay to be parameterized. This #1**step** construct is a conceptual mechanism that provides a method for defining when sampling takes place and does not require that an event be created in this previous time slot. Conceptually, this #1**step** sampling is identical to taking the data samples in the Preponed region of the current time slot.

The code specified in the program block and the pass/fail code from property expressions are scheduled in the Reactive region. A #0 control delay specified in a program block schedules the process for resumption in the Re-inactive region. The Re-inactive region is the program block dual of the Inactive region (see below).

The Pre-active, Pre-NBA, and Post-NBA regions are new in this standard but support existing PLI call-backs. The Post-observed region is new in this standard and has been added for PLI support.

The Pre-active region provides for a PLI callback control point that allows PLI application routines to read and write values and create events before events in the Active region are evaluated (see 9.4).

The Pre-NBA region provides for a PLI callback control point that allows PLI application routines to read and write values and create events before the events in the NBA region are evaluated (see 9.4).

The Post-NBA region provides for a PLI callback control point that allows PLI application routines to read and write values and create events after the events in the NBA region are evaluated (see 9.4).

The Post-observed region provides for a PLI callback control point that allows PLI application routines to read values after properties are evaluated (in Observed or earlier region).

NOTE—The PLI currently does not schedule callbacks in the Post-observed region.

The Pre-postponed region provides a PLI callback control point that allows PLI application routines to read and write values and create events after processing all other regions except the Postponed region.

The flow of execution of the event regions is specified in Figure 9-1.

The Active, Inactive, Pre-NBA, NBA, Post-NBA, Observed, Post-observed, Reactive, Re-inactive, and Pre-postponed regions are known as the *iterative* regions.

The Preponed region provides for a PLI callback control point that allows PLI application routines to access data at the current time slot before any net or variable has changed state. Within this region, it is illegal to write values to any net or variable or to schedule an event in any other region within the current time slot.

NOTE—The PLI currently does not schedule callbacks in the Preponed region.

The Active region holds current events being evaluated and can be processed in any order.

The Inactive region holds the events to be evaluated after all the active events are processed.

An *explicit* #0 delay control requires that the process be suspended and an event scheduled into the Inactive region (or Re-inactive for program blocks) of the current time slot so that the process can be resumed in the next inactive to active iteration.

A nonblocking assignment creates an event in the NBA region, scheduled for the current or a later simulation time.

The Postponed region provides for a PLI callback control point that allows PLI application routines to be suspended until after all the Active, Inactive, NBA, Observed, Reactive, and Re-inactive regions have completed. Within this region, it is illegal to write values to any net or variable or to schedule an event in any previous region within the current time slot.
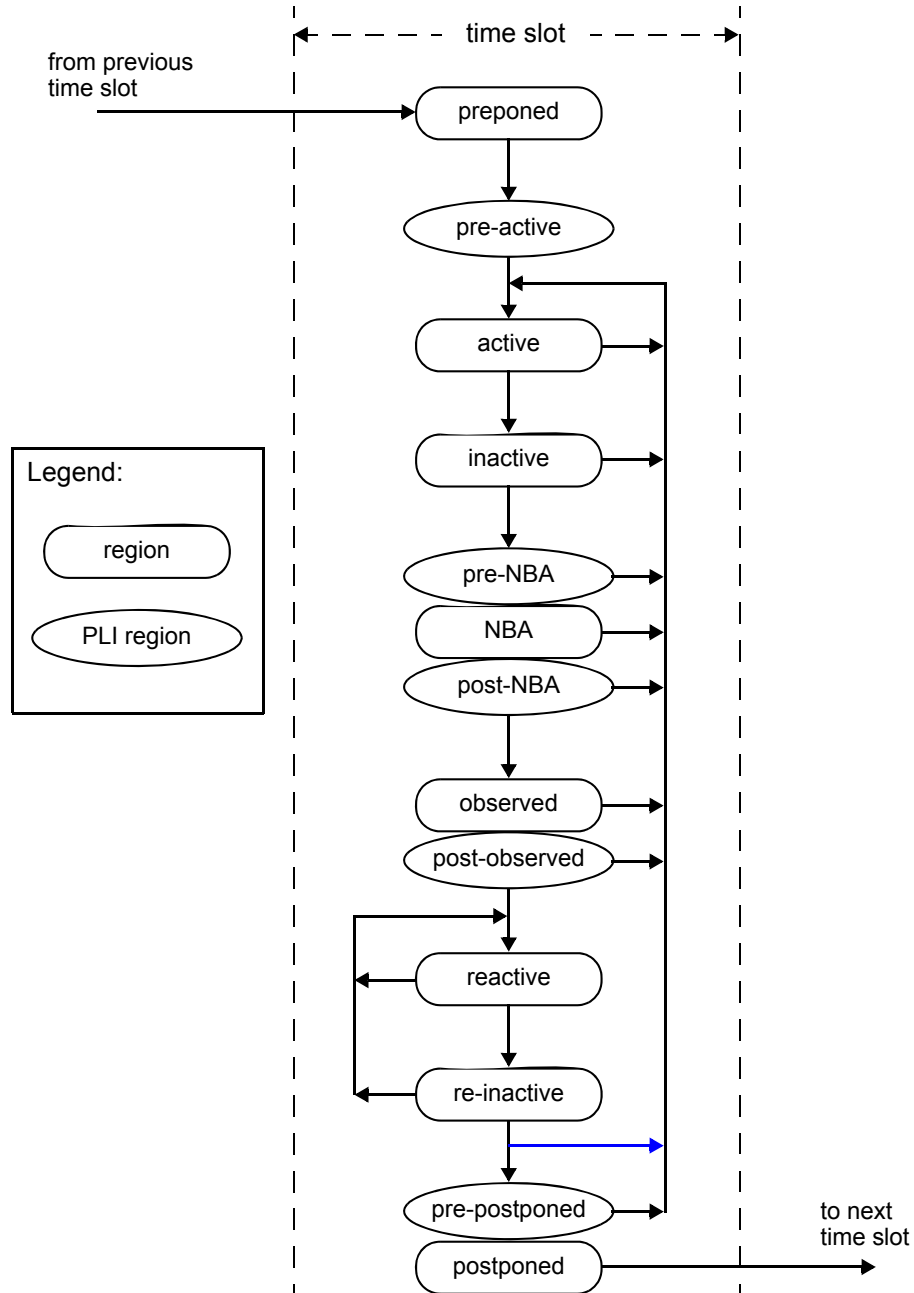
Figure 9-1—The SystemVerilog flow of time slots and event regions

9.3.1 The SystemVerilog simulation reference algorithm

```
execute_simulation {
    T = 0;
    initialize the values of all nets and variables;
    schedule all initialization events into time 0 slot;
    while (some time slot is nonempty) {
        move to the next future nonempty time slot and set T;
```

```
        execute_time_slot (T);
    }
}

execute_time_slot {
    execute_region (preponed);
    execute_region (pre-active);
    while (any region in [active ... pre-postponed] is nonempty) {
        while (any region in [active ... post-observed] is nonempty) {
            execute_region (active);
            R = first nonempty region in [active ... post-observed];
            if (R is nonempty)
                move events in R to the active region;
        }
        while (any region in [reactive ... re-inactive] is nonempty) {
            execute_region (reactive);
            R = first nonempty region in [reactive ... re-inactive];
            if (R is nonempty)
                move events in R to the reactive region;
        }
        if (all regions in [active … re-inactive] are empty)
            execute_region (pre-postponed);
    }
    execute_region (postponed);
}

execute_region {
    while (region is nonempty) {
        E = any event from region;
        remove E from the region;
        if (E is an update event) {
            update the modified object;
            evaluate processes sensitive to the object and possibly schedule
                further events for execution;
        } else { /* E is an evaluation event */
            evaluate the process associated with the event and possibly
                schedule further events for execution;
        }
    }
}
```

The Iterative regions and their order are Active, Inactive, Pre-NBA, NBA, Post-NBA, Observed, Post-observed, Reactive, Re-inactive, and Pre-postponed. As shown in the algorithm, once the Reactive or Re-Inactive regions are processed, iteration over the other regions does not resume until these two regions are empty.

9.4 The PLI callback control points

There are two kinds of PLI callbacks: those that are executed immediately when some specific activity occurs and those that are explicitly registered as a one-shot evaluation event.

Table 9-1 provides the mapping from the various PLI callbacks.

Table 9-1—PLI callbacks

Callback	Event region
`cbAfterDelay`	Pre-active
`cbNextSimTime`	Pre-active
`cbReadWriteSynch`	Pre-NBA or Post-NBA
`cbAtStartOfSimTime`	Pre-active
`cbNBASynch`	Pre-NBA
`cbAtEndOfSimTime`	Pre-postponed
`cbReadOnlySynch`	Postponed

10. Procedural statements and control flow

10.1 Introduction

NOTE—Procedural statements are introduced by the following:

initial // enable this statement at the beginning of simulation and execute it only once

final // do this statement once at the end of simulation

always, **always_comb**, **always_latch**, **always_ff** // loop forever (see Clause 11 on processes)

task // do these statements whenever the task is called

function // do these statements whenever the function is called and return a value

SystemVerilog has the following types of control flow within a process:

— Selection, loops, and jumps
— Task and function calls
— Sequential and parallel blocks
— Timing control

Verilog procedural statements are in **initial** or **always** blocks, tasks, or functions. SystemVerilog adds a **final** block that executes at the end of simulation.

Verilog includes most of the statement types of C, except for do...while, break, continue, return, and goto. Verilog has the **repeat** statement, which C does not, and the **disable**. The use of the Verilog **disable** to carry out the functionality of break and continue requires the user to invent block names and introduces the opportunity for error.

SystemVerilog adds C-like **break**, **continue**, and **return** functionality, which do not require the use of block names.

Loops with a test at the end are sometimes useful to save duplication of the loop body. SystemVerilog adds a C-like **do**...**while** loop for this capability.

Verilog provides two overlapping methods for procedurally adding and removing drivers for variables: the **force**/**release** statements and the **assign**/**deassign** statements. The **force**/**release** statements can also be used to add or remove drivers for nets in addition to variables. A **force** statement targeting a variable that is currently the target of an assign shall override that assign; however, once the force is released, the assign's effect again shall be visible.

The keyword **assign** is much more commonly used for the somewhat similar, yet quite different, purpose of defining permanent drivers of values to nets.

SystemVerilog **final** blocks execute in an arbitrary but deterministic sequential order. This is possible because **final** blocks are limited to the legal set of statements allowed for functions. SystemVerilog does not specify the ordering, but implementations should define rules that preserve the ordering between runs. This helps keep the output log file stable because **final** blocks are mainly used for displaying statistics.

10.2 Statements

The syntax for procedural statements is as follows:

```
statement_or_null ::=                                                    // from A.6.4
        statement
      | { attribute_instance } ;
statement ::= [ block_identifier : ] { attribute_instance } statement_item
statement_item ::=
        blocking_assignment ;
      | nonblocking_assignment ;
      | procedural_continuous_assignment ;
      | case_statement
      | conditional_statement
      | inc_or_dec_expression ;
      | subroutine_call_statement
      | disable_statement
      | event_trigger
      | loop_statement
      | jump_statement
      | par_block
      | procedural_timing_control_statement
      | seq_block
      | wait_statement
      | procedural_assertion_statement
      | clocking_drive ;
      | randsequence_statement
      | randcase_statement
      | expect_property_statement
```

Syntax 10-1—Procedural statement syntax (excerpt from Annex A)

10.3 Blocking and nonblocking assignments

```
blocking_assignment ::=                                                  // from A.6.2
        variable_lvalue = delay_or_event_control  expression
      | hierarchical_dynamic_array_variable_identifier = dynamic_array_new
      | [ implicit_class_handle . | class_scope | package_scope ] hierarchical_variable_identifier
            select = class_new
      | operator_assignment
operator_assignment ::= variable_lvalue  assignment_operator  expression
assignment_operator ::=
        = | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>= | <<<= | >>>=
nonblocking_assignment ::= variable_lvalue <= [ delay_or_event_control ] expression
```

Syntax 10-2—Blocking and nonblocking assignment syntax (excerpt from Annex A)

The following assignments are allowed in both Verilog and SystemVerilog:

```
#1 r = a;
r = #1 a;
r <= #1 a;
r <= a;
@c r = a;
```

```
    r = @c a;
    r <= @c a;
```

SystemVerilog also allows a time unit to be specified in the assignment statement, as follows:

```
    #1ns r = a;
    r = #1ns a;
    r <= #1ns a;
```

It shall be illegal to make nonblocking assignments to automatic variables.

The size of the left-hand side of an assignment forms the context for the right-hand expression. If the left-hand side is smaller than the right-hand side, information can be lost, and a warning can be given.

10.4 Selection statements

---

conditional_statement ::=   *// from A.6.6*
      if (cond_predicate **)** statement_or_null [**else** statement_or_null]
    | unique_priority_if_statement
unique_priority_if_statement ::=
      [unique_priority] **if (** cond_predicate **)** statement_or_null
        { **else if (** cond_predicate **)** statement_or_null }
        [**else** statement_or_null]
unique_priority ::= **unique** | **priority**
cond_predicate ::=
      expression_or_cond_pattern { **&&&** expression_or_cond_pattern }
expression_or_cond_pattern ::=
      expression | cond_pattern
cond_pattern ::= expression **matches** pattern
case_statement ::=   *// from A.6.7*
      [unique_priority] case_keyword **(** expression **)** case_item { case_item } **endcase**
    | [unique_priority] case_keyword **(** expression **) matches** case_pattern_item { case_pattern_item }
        endcase
    | [unique_priority] **case (** expression **) inside** case_inside_item { case_inside_item } **endcase**
case_keyword ::= **case** | **casez** | **casex**
case_item ::=
      expression { **,** expression } **:** statement_or_null
    | **default** [**:**] statement_or_null
case_pattern_item ::=
      pattern [**&&&** expression] **:** statement_or_null
    | **default** [**:**] statement_or_null
case_inside_item ::=
      open_range_list **:** statement_or_null
    | **default** [**:**] statement_or_null

Syntax 10-3—Selection statement syntax (excerpt from Annex A)

In Verilog, an **if** (*expression*) is evaluated as a boolean so that if the result of the expression is 0 or X, the test is considered false.

SystemVerilog adds the keywords **unique** and **priority**, which can be used before an **if**. If either keyword is used, it shall be a warning for no condition to match unless there is an explicit **else**. For example:

```
unique if ((a==0) || (a==1)) $display("0 or 1");
else if (a == 2) $display("2");
else if (a == 4) $display("4"); // values 3,5,6,7 cause a warning

priority if (a[2:1]==0) $display("0 or 1");
else if (a[2] == 0) $display("2 or 3");
else $display("4 to 7"); //covers all other possible values, so no warning
```

A **unique if** asserts that there is no overlap in a series of **if**...**else**...**if** conditions, i.e., they are mutually exclusive and hence it is safe for the expressions to be evaluated in parallel. In a **unique if**, it shall be legal for a condition to be evaluated at any time after entrance into the series and before the value of the condition is needed. A **unique if** shall be illegal if, for any such interleaving of evaluation and use of the conditions, more than one condition is true. For an illegal **unique if**, an implementation shall be required to issue a warning, unless it can demonstrate a legal interleaving so that no more than one condition is true.

To implement this requirement, an implementation can continue the evaluation of conditions after a first true condition has been found and even after the execution of the statement associated with the first true condition. However, the statements associated with any additional true conditions shall not be executed. An implementation shall also issue a warning if it determines that no condition is true, or it is possible that no condition is true, and the final **if** does not have a corresponding **else**.

A **priority if** indicates that a series of **if**...**else**...**if** conditions shall be evaluated in the order listed. In the preceding example, if the variable a had a value of 0, it would satisfy both the first and second conditions, requiring priority logic. An implementation shall also issue a warning if it determines that no condition is true, or it is possible that no condition is true, and the final **if** does not have a corresponding **else**.

The **unique** and **priority** keywords apply to the entire series of **if**...**else**...**if** conditions. In the preceding examples, it would have been illegal to insert either keyword after any of the occurrences of **else**. To nest another **if** statement within such a series of conditions, a **begin**...**end** block should be used.

In Verilog, there are three types of **case** statements, introduced by **case**, **casez**, and **casex**. With System-Verilog, each of these can be qualified by **priority** or **unique**. A **priority case** shall act on the first match only. A **unique case** asserts that there are no overlapping case items and hence that it is safe for the case items to be evaluated in parallel. In a **unique case**, it shall be legal to evaluate a case item expression at any time after the evaluation of the case expression and before the evaluation of the corresponding comparison. A **unique case** shall be illegal if, for any such interleaving of evaluations and comparisons, more than one case item matches the case expression. For an illegal **unique case**, an implementation shall be required to issue a warning message, unless it can demonstrate a legal interleaving of evaluations and comparisons so that no more than one case item matches the case expression. To implement this requirement, an implementation can continue the evaluations and comparisons after the termination of the usual linear search and even after the execution of the statement associated with the first matching case item. However, the statements associated with any additional matching case items shall not be executed.

If the case is qualified as **priority** or **unique**, the simulator shall issue a warning message if no case item matches. These warnings can be issued at either compile time or run time, as soon as it is possible to determine the illegal condition.

NOTE—By specifying **unique** or **priority**, it is not necessary to code a **default** case to trap unexpected case values.

Consider the following example:

```
bit [2:0] a;
unique case(a) // values 3,5,6,7 cause a warning
      0,1: $display("0 or 1");
      2: $display("2");
      4: $display("4");
endcase

priority casez(a) // values 4,5,6,7 cause a warning
      3'b00?: $display("0 or 1");
      3'b0??: $display("2 or 3");
endcase
```

The keyword **inside** can be used after the parenthesized expression to indicate a set membership (see 8.19) **case**...**inside** statement. In a **case**...**inside** statement, the case expression shall be compared with each case item expression (*open_range_list*) using the set membership **inside** operator. The **inside** operator uses asymmetric wild card matching (see 8.5). Accordingly, the case expression shall be the left operand, and each case item expression shall be the right operand. The case expression given in parentheses and each case item expression in braces shall be evaluated in the order specified by a normal **case**, **unique case**, or **priority case** statement. A case item shall be matched when the **inside** operation compares the case expression to the case item expression and returns 1'b1 and no match when the operation returns 1'b0 or 1'bx. If all comparisons do not match and the default item is given, the default item statement shall be executed.

For example:

```
logic [2:0] status;
always @(posedge clock)
   priority case (status) inside
   1, 3 : task1; // matches b001 and b011
   3'b0?0, [4:7]: task2; // matches b000 b010 b0x0 b0z0 b100 b101 b110 b111
   endcase // priority case fails all other values including b00x b01x bxxx
```

10.4.1 Pattern matching

Pattern matching provides a visual and succinct notation to compare a value against structures, tagged unions, and constants and to access their members. SystemVerilog adds pattern matching capability to **case** and **if** statements and to conditional expressions. Before describing pattern matching in those contexts, the general concepts are described first.

A pattern is a nesting of tagged union and structure expressions with identifiers, constant expressions, and the wildcard pattern ".*" at the leaves. For tagged union patterns, the identifier following the **tagged** keyword is a union member name. For **void** members, the nested member pattern is omitted.

```
pattern ::=                                                        // from A.6.7.1
      . variable_identifier
    | .*
    | constant_expression
    | tagged member_identifier [ pattern ]
    | '{ pattern { , pattern } }
    | '{ member_identifier : pattern { , member_identifier : pattern } }
```

Syntax 10-4—Pattern syntax (excerpt from Annex A)

A pattern always occurs in a context of known type because it is matched against an expression of known type. Recursively, its nested patterns also have known type. A constant expression pattern must be of integral type. Thus a pattern can always be statically type-checked.

Each pattern introduces a new scope; the extent of this scope is described separately below for **case** statements, **if** statements, and conditional expressions. Each pattern identifier is implicitly declared as a new variable within the pattern's scope, with the unique type that it must have based on its position in the pattern. Pattern identifiers must be unique in the pattern, i.e., the same identifier cannot be used in more than one position in a single pattern.

In pattern-matching, the value V of an expression is always matched against a pattern, and static type checking ensures that V and the pattern have the same type. The result of a pattern match is as follows:

— A 1-bit determined value: 0 (false, or fail) or 1 (true, or succeed). The result cannot be x or z even if the value and pattern contain such bits.

— If the match succeeds, the pattern identifiers are bound to the corresponding members from V, using ordinary procedural assignment.

— Each pattern is matched using the following simple recursive rule:

  — An identifier pattern always succeeds (matches any value), and the identifier is bound to that value (using ordinary procedural assignment).

  — The wildcard pattern ".*" always succeeds.

  — A constant expression pattern succeeds if V is equal to its value.

  — A tagged union pattern succeeds if the value has the same tag and, recursively, if the nested pattern matches the member value of the tagged union.

  — A structure pattern succeeds if, recursively, each of the nested member patterns matches the corresponding member values in V. In structure patterns with named members, the textual order of members does not matter, and members can be omitted. Omitted members are ignored.

Conceptually, if the value V is seen as a flattened vector of bits, the pattern specifies which bits to match, with what values they should be matched, and, if the match is successful, which bits to extract and bind to the pattern identifiers. Matching can be insensitive to x and z values, as described in the individual constructs below.

10.4.1.1 Pattern matching in case statements

In a pattern-matching **case** statement, the expression in parentheses is followed by the keyword **matches**, and the statement contains a series of "case_pattern_items". The left-hand side of a case item, before the ":", consists of a pattern and, optionally, the operator **&&&** followed by a boolean "filter" expression. The right-hand side of a case item is a statement. Each pattern introduces a new scope, in which its pattern identifiers are implicitly declared; this scope extends to the optional filter expression and the statement in the right-hand side of the same case item. Thus different case items can reuse pattern identifiers.

All the patterns are completely statically type-checked. The expression being tested in the pattern-matching **case** statement must have a known type, which is the same as the type of the pattern in each case item.

The expression in parentheses in a pattern-matching **case** statement shall be evaluated exactly once. Its value V shall be matched against the left-hand sides of the case items, one at a time, in the exact order given, ignoring the default case item if any. During this linear search, if a case item is selected, its statement is executed and the linear search is terminated. If no case item is selected and a default case item is given, then its statement is executed. If no case item is selected and no default case item is given, no statement is executed.

For a case item to be selected, the value *V* must match the pattern (and the pattern identifiers are assigned the corresponding member values in *V*), and then the boolean filter expression must evaluate to true (a determined value other than 0).

Example:

```
typedef union tagged {
   void Invalid;
   int Valid;
} VInt;
...
VInt v;
...
case (v) matches
   tagged Invalid  : $display ("v is Invalid");
   tagged Valid .n : $display ("v is Valid with value %d", n);
endcase
```

In the **case** statement, if v currently has the Invalid tag, the first pattern is matched. Otherwise, it must have the Valid tag, and the second pattern is matched. The identifier n is bound to the value of the Valid member, and this value is displayed. It is impossible to access the integer member value (n) when the tag is Invalid.

Example:

```
typedef union tagged {
   struct {
      bit [4:0] reg1, reg2, regd;
   } Add;
   union tagged {
      bit [9:0] JmpU;
      struct {
         bit [1:0] cc;
         bit [9:0] addr;
      } JmpC;
   } Jmp;
} Instr;
...
Instr instr;
...
case (instr) matches
   tagged Add '{.r1, .r2, .rd} &&& (rd != 0): rf[rd] = rf[r1] + rf[r2];
   tagged Jmp .j : case (j) matches
                     tagged JmpU .a       : pc = pc + a;
                     tagged JmpC '{.c, .a}: if (rf[c]) pc = a;
                   endcase
endcase
```

If instr holds an Add instruction, the first pattern is matched, and the identifiers r1, r2, and rd are bound to the three register fields in the nested structure value. The right-hand statement executes the instruction on the register file rf. It is impossible to access these register fields if the tag is Jmp. If instr holds a Jmp instruction, the second pattern is matched, and the identifier j is bound to the nested tagged union value. The inner **case** statement, in turn, matches this value against JmpU and JmpC patterns and so on.

Example (same as previous example, but using wildcard and constant patterns to eliminate the rd = 0 case; in some processors, register 0 is a special "discard" register):

```
    case (instr) matches
        tagged Add '{.*, .*, 0}    : ; // no op
        tagged Add '{.r1, .r2, .rd}  : rf[rd] = rf[r1] + rf[r2];
        tagged Jmp .j : case (j) matches
                            tagged JmpU .a        : pc = pc + a;
                            tagged JmpC '{.c, .a} : if (rf[c]) pc = a;
                        endcase
    endcase
```

Example (same as previous example except that the first inner **case** statement involves only structures and constants but no tagged unions):

```
    case (instr) matches
        tagged Add s:  case (s) matches
                            '{.*, .*, 0}    : ; // no op
                            '{.r1, .r2, .rd}  : rf[rd] = rf[r1] + rf[r2];
                        endcase
        tagged Jmp .j: case (j) matches
                            tagged JmpU .a        : pc = pc + a;
                            tagged JmpC '{.c, .a} : if (rf[c]) pc = a;
                        endcase
    endcase
```

Example (same as previous example, but using nested tagged union patterns):

```
    case (instr) matches
        tagged Add '{.r1, .r2, .rd} &&& (rd != 0) : rf[rd] = rf[r1] + rf[r2];
        tagged Jmp (tagged JmpU .a)                : pc = pc + a;
        tagged Jmp (tagged JmpC '{.c, .a})         : if (rf[c]) pc = a;
    endcase
```

Example (same as previous example, with named structure components):

```
    case (instr) matches
        tagged Add '{reg2:.r2,regd:.rd,reg1:.r1} &&& (rd != 0):
                                              rf[rd] = rf[r1] + rf[r2];
        tagged Jmp (tagged JmpU .a)                : pc = pc + a;
        tagged Jmp (tagged JmpC '{addr:.a,cc:.c}) : if (rf[c]) pc = a;
    endcase
```

As usual, the **casez** and **casex** keywords can be used instead of **case**, with the same semantics. In other words, during pattern matching, wherever 2 bits are compared (whether they are tag bits or members), the **casez** form ignores z bits, and the **casex** form ignores both z and x bits.

The **priority** and **unique** qualifiers play their usual role. **priority** implies that some case item must be selected. **unique** also implies that exactly one case item will be selected so that they can be evaluated in parallel.

10.4.1.2 Pattern matching in if statements

The predicate in an **if** statement can be a series of clauses separated with the $\&\&\&$ operator. Each clause is either an expression (used as a boolean filter) or has the form expression matches pattern. The clauses represent a sequential conjunction from left to right, i.e., if any clause fails, the remaining clauses are not evaluated, and all of them must succeed for the predicate to be true. Boolean expression clauses are evaluated as usual. Each pattern introduces a new scope, in which its pattern identifiers are implicitly declared; this scope extends to the remaining clauses in the predicate and to the corresponding "true" arm of the **if** statement.

In each *e* **matches** *p* clause, *e* and *p* must have the same known statically known type. The value of *e* is matched against the pattern *p* as described above.

Even though the pattern-matching clauses always return a defined 1-bit result, the overall result can be ambiguous because of the boolean filter expressions in the predicate. The standard semantics of **if** statements holds, i.e., the first statement is executed if, and only if, the result is a determined value other than 0.

Example:

```
if (e matches (tagged Jmp (tagged JmpC '{cc:.c,addr:.a})))
   ... // c and a can be used here
else
...
```

Example (same as previous example, illustrating a sequence of two pattern matches with identifiers bound in the first pattern used in the second pattern).

```
if (e matches (tagged Jmp .j),
    j matches (tagged JmpC '{cc:.c,addr:.a}))
      ... // c and a can be used here
else
...
```

Example (same as first example, but adding a boolean expression to the sequence of clauses). The idea expressed is "if e is a conditional jump instruction and the condition register is not equal to zero ...".

```
if (e matches (tagged Jmp (tagged JmpC '{cc:.c,addr:.a}))
   &&& (rf[c] != 0))
   ... // c and a can be used here
else
...
```

The **priority** and **unique** qualifiers play their usual role for **if** statements even if they use pattern matching.

10.4.1.3 Pattern matching in conditional expressions

A conditional expression (e1 ? e2 : e3) can also use pattern matching, i.e., the predicate e1 can be a sequence of expressions and "expression **matches** pattern" clauses separated with the &&& operator, just like the predicate of an **if** statement. The clauses represent a sequential conjunction from left to right, i.e., if any clause fails, the remaining clauses are not evaluated, and all of them must succeed for the predicate to be true. Boolean expression clauses are evaluated as usual. Each pattern introduces a new scope, in which its pattern identifiers are implicitly declared; this scope extends to the remaining clauses in the predicate and to the consequent expression e2.

As described in the previous subclause, e1 can evaluate to true, false, or an ambiguous value. The semantics of the overall conditional expression is described in 8.17, based on these three possible outcomes for e1.

10.5 Loop statements

```
loop_statement ::=                                                           // from A.6.8
        forever statement_or_null
        | repeat ( expression ) statement_or_null
        | while ( expression ) statement_or_null
        | for ( for_initialization ; expression ; for_step )
            statement_or_null
        | do statement_or_null while ( expression ) ;
        | foreach ( array_identifier [ loop_variables ] ) statement
for_initialization ::=
        list_of_variable_assignments
        | for_variable_declaration { , for_variable_declaration }
for_variable_declaration ::=
        data_type variable_identifier = expression { , variable_identifier = expression }
for_step ::= for_step_assignment { , for_step_assignment }
for_step_assignment ::=
        operator_assignment
        | inc_or_dec_expression
        | function_subroutine_call
loop_variables ::= [ index_variable_identifier ] { , [ index_variable_identifier ] }
```

Syntax 10-5—Loop statement syntax (excerpt from Annex A)

Verilog provides **for**, **while**, **repeat** and **forever** loops. SystemVerilog enhances the Verilog **for** loop, and adds a **do**...**while** loop and a **foreach** loop.

10.5.1 The do...while loop

```
do statement while(condition) // as C
```

The condition can be any expression that can be treated as a boolean. It is evaluated after the statement.

10.5.2 Enhanced for loop

In Verilog, the variable used to control a **for** loop must be declared prior to the loop. If loops in two or more parallel procedures use the same loop control variable, there is a potential of one loop modifying the variable while other loops are still using it.

SystemVerilog adds the ability to declare the **for** loop control variable within the **for** loop. This creates a local variable within the loop. Other parallel loops cannot inadvertently affect the loop control variable. For example:

```
module foo;

    initial begin
        for (int i = 0; i <= 255; i++)
            ...
    end

    initial begin
        loop2: for (int i = 15; i >= 0; i--)
```

```
          ...
      end
  endmodule
```

The local variable declared within a **for** loop is equivalent to declaring an automatic variable in an unnamed block.

Verilog only permits a single initial statement and a single step assignment within a **for** loop. SystemVerilog allows the initial declaration or assignment statement to be one or more comma-separated statements. The step assignment can also be one or more comma-separated assignment statements.

```
for ( int count = 0; count < 3; count++ )
   value = value +((a[count]) * (count+1));

for ( int count = 0, done = 0, j = 0; j * count < 125; j++, count++)
   $display("Value j = %d\n", j );
```

In a **for** loop initialization, either all or none of the control variables are locally declared. In the second loop of the last example, count, done, and j are all locally declared. The following would be illegal because it is attempting to locally declare y whereas x was not locally declared:

```
for (x = 0, int y = 0; ...)
...
```

In a **for** loop initialization that declares multiple local variables, the initialization expression of a local variable can use earlier local variables.

```
for (int i = 0, j = i+offset; i < N; i++,j++)
...
```

10.5.3 The foreach loop

The **foreach** construct specifies iteration over the elements of an array. Its argument is an identifier that designates any type of array (fixed-size, dynamic, or associative) followed by a list of loop variables enclosed in square brackets. Each loop variable corresponds to one of the dimensions of the array. The **foreach** construct is similar to a repeat loop that uses the array bounds to specify the repeat count instead of an expression.

Examples:

```
string words [2] = '{ "hello", "world" };
int prod [1:8] [1:3];

foreach( words [ j ] )
   $display( j , words[j] );      // print each index and value

foreach( prod[ k, m ] )
   prod[k][m] = k * m;            // initialize
```

The number of loop variables must match the number of dimensions of the array variable. Empty loop variables can be used to indicate no iteration over that dimension of the array, and contiguous empty loop variables towards the end can be omitted. Loop variables are automatic and read-only, and their scope is local to the loop. The type of each loop variable is implicitly declared to be consistent with the type of array index. It shall be an error for any loop variable to have the same identifier as the array.

The mapping of loop variables to array indexes is determined by the dimension cardinality, as described in 22.6. The **foreach** arranges for higher cardinality indexes to change more rapidly.

```
//    1 2 3            3   4       1   2   -> Dimension numbers
int A [2][3][4];    bit [3:0][2:1] B [5:1][4];

foreach( A [ i, j, k ] ) ...
foreach( B [ q, r, , s ] ) ...
```

The first **foreach** causes i to iterate from 0 to 1, j from 0 to 2, and k from 0 to 3. The second **foreach** causes q to iterate from 5 to 1, r from 0 to 3, and s from 2 to 1 (iteration over the third index is skipped).

Multiple loop variables correspond to nested loops that iterate over the given indexes. The nesting of the loops is determined by the dimension cardinality; outer loops correspond to lower cardinality indexes. In the first example above, the outermost loop iterates over i, and the innermost loop iterates over k.

When loop variables are used in expressions other than as indexes to the designated array, they are auto-cast into a type consistent with the type of index. For fixed-size and dynamic arrays, the auto-cast type is **int**. For associative arrays indexed by a specific index type, the auto-cast type is the same as the index type. For associative arrays indexed by a wildcard index (*****), the auto-cast type is **longint unsigned**. To use different types, an explicit cast can be used.

10.6 Jump statements

---
jump_statement ::=   *// from A.6.5*
      return [expression] **;**
   | **break ;**
   | **continue ;**

Syntax 10-6—Jump statement syntax (excerpt from Annex A)

SystemVerilog adds the C jump statements **break**, **continue**, and **return**.

```
break              // out of loop as C
continue           // skip to end of loop as C
return expression  // exit from a function
return             // exit from a task or void function
```

The **continue** and **break** statements can only be used in a loop. The **continue** statement jumps to the end of the loop and executes the loop control if present. The **break** statement jumps out of the loop. The **continue** and **break** statements cannot be used inside a **fork**...**join** block to control a loop outside the **fork**...**join** block.

The **return** statement can only be used in a task or function. In a function returning a value, the return must have an expression of the correct type.

NOTE—SystemVerilog does not include the C **goto** statement.

10.7 Final blocks

The **final** block is like an **initial** block, defining a procedural block of statements, except that it occurs at the end of simulation time and executes without delays. A **final** block is typically used to display statistical information about the simulation.

final_construct ::= **final** function_statement	*// from A.6.2*

Syntax 10-7—Final block syntax (excerpt from Annex A)

The only statements allowed inside a **final** block are those permitted inside a function declaration. This guarantees that they execute within a single simulation cycle. Unlike an **initial** block, the **final** block does not execute as a separate process; instead, it executes in zero time, the same as a function call.

A **final** block executes when simulation ends due to an explicit or implicit call to $finish.

```
final
   begin
      $display("Number of cycles executed %d",$time/period);
      $display("Final PC = %h",PC);
   end
```

Execution of $finish, tf_dofinish(), or vpi_control(vpiFinish,...) from within a **final** block shall cause the simulation to end immediately. A **final** block can only trigger once in a simulation.

A **final** block shall execute before any PLI callbacks that indicate the end of simulation.

10.8 Named blocks and statement labels

seq_block ::=	*// from A.6.3*
begin [**:** block_identifier] { block_item_declaration } { statement_or_null }	
end [**:** block_identifier]	
par_block ::=	
fork [**:** block_identifier] { block_item_declaration } { statement_or_null }	
join_keyword [**:** block_identifier]	
join_keyword ::= **join** | **join_any** | **join_none**	

Syntax 10-8—Blocks and labels syntax (excerpt from Annex A)

Verilog allows a **begin**...**end**, **fork**...**join**, **fork**...**join_any**, or **fork**...**join_none** statement block to be named. A named block is used to identify the entire statement block. A named block creates a new hierarchy scope. The block name is specified after the **begin** or **fork** keyword, preceded by a colon. For example:

```
begin : blockA    // Verilog named block
   ...
end
```

SystemVerilog allows a matching block name to be specified after the block **end**, **join**, **join_any**, or **join_none** keyword, preceded by a colon. This can help document which **end** or **join**, **join_any**, or **join_none** is associated with which **begin** or **fork** when there are nested blocks. A name at the end of the block is not required. It shall be an error if the name at the end is different from the block name at the beginning.

```
begin: blockB     // block name after the begin or fork
   ...
end: blockB
```

SystemVerilog allows a label to be specified before any statement, as in C. A statement label is used to iden-
tify a single statement. The label name is specified before the statement, followed by a colon.

```
labelA: statement
```

A **begin**...**end**, **fork**...**join**, **fork**...**join_any**, or **fork**...**join_none** block is considered a statement and
can have a statement label before the block.

```
labelB: fork   // label before the begin or fork
   ...
join : labelB
```

It shall be illegal to have both a label before a **begin** or **fork** and a block name after the **begin** or **fork**. A
label cannot appear before the **end**, **join**, **join_any**, or **join_none**, as these keywords do not form a
statement.

A statement with a label can be disabled using a **disable** statement. Disabling a statement shall have the
same behavior as disabling a named block.

See 11.6 for additional discussion on **fork**...**join**, **fork**...**join_any**, or **fork**...**join_none**.

10.9 Disable

SystemVerilog has **break** and **continue** to break out of or continue the execution of loops. The Verilog
disable can also be used to break out of or continue a loop, but is more awkward than using **break** or
continue. The **disable** is also allowed to disable a named block, which does not contain the **disable**
statement. If the block is currently executing, this causes control to jump to the statement immediately after
the block. If the block is a loop body, it acts like a **continue**. If the block is not currently executing, the
disable has no effect.

SystemVerilog has **return** from a task, but **disable** is also supported. If **disable** is applied to a named
task, all current executions of the task are disabled.

```
module ...
always always1: begin ... t1: task1( ); ... end
...
endmodule

always begin
   ...
   disable u1.always1.t1; // exit task1, which was called from always1
(static)
end
```

10.10 Event control

```
delay_or_event_control ::=                                          // from A.6.5
        delay_control
      | event_control
      | repeat ( expression ) event_control
delay_control ::=
        # delay_value
      | # ( mintypmax_expression )
event_control ::=
        @ hierarchical_event_identifier
      | @ ( event_expression )
      | @*
      | @ (*)
      | @ sequence_instance
event_expression ::=
        [ edge_identifier ] expression [ iff expression ]
      | sequence_instance [ iff expression ]
      | event_expression or event_expression
      | event_expression , event_expression
edge_identifier ::= posedge | negedge                              // from A.7.4
```

Syntax 10-9—Delay and event control syntax (excerpt from Annex A)

Any change in a variable or net can be detected using the `@` event control, as in Verilog. If the expression evaluates to a result of more than 1 bit, a change on any of the bits of the result (including an x to z change) shall trigger the event control.

SystemVerilog adds an `iff` qualifier to the `@` event control.

```
    module latch (output logic [31:0] y, input [31:0] a, input enable);
       always @(a iff enable == 1)
          y <= a; //latch is in transparent mode
    endmodule
```

The event expression only triggers if the expression after the `iff` is true, in this case when `enable` is equal to 1. This type of expression is evaluated when `a` changes and not when `enable` changes. Also, in similar event expressions of this type, `iff` has precedence over `or`. This can be made clearer by the use of parentheses.

If a variable is not of a 4-state type, then `posedge` and `negedge` refer to transitions from 0 and to 0, respectively.

If the expression denotes a `clocking` block `input` or `inout` (see Clause 15), the event control operator uses the synchronous values, that is, the values sampled by the clocking event. The expression can also denote a `clocking` block name (with no edge qualifier) to be triggered by the clocking event.

A variable used with the event control can be any one of the integral data types (see 4.3.1) or string. The variable can be either a simple variable or a `ref` argument (variable passed by reference); it can be a member of an array, associative-array, or object (class instance) of the aforementioned types.

Event expressions must return singular values. Aggregate types can be used in an expression provided the expression reduces to a singular value. The object members or aggregate elements can be any type as long as the result of the expression is a singular value.

If the event expression is a reference to a simple object handle or chandle variable, an event is created when a write to that variable is not equal to its previous value.

Nonvirtual methods of an object and built-in methods or system functions for an aggregate type are allowed in event control expressions as long as the type of the return value is singular and the method is defined as a function, not a task.

Changing the value of object data members, aggregate elements, or the size of a dynamically sized array referenced by a method or function shall cause the event expression to be reevaluated. An implementation can cause the event expression to be reevaluated when changing the value or size even if the members are not referenced by the method or function.

```
real AOR[];                          // dynamic array of reals
byte stream[$];                      // queue of bytes
initial wait(AOR.size() > 0) ....;   // waits for array to be allocated
initial wait($bits(stream) > 60)...; // waits for total number of bits
                                     // in stream greater than 60

Packet p = new; // Packet 1
Packet q = new; // Packet 2
initial fork
   @(p.status);   // Wait for status in Packet 1 to change
   @ q;           // Wait for a change to handle q
   # 10 q = p;    // triggers @q.
   // @(p.status) now waits for status in Packet 2 to change,
   // if not already different from Packet 1
join
```

10.10.1 Sequence events

A sequence instance can be used in event expressions to control the execution of procedural statements based on the successful match of the sequence. This allows the end point of a named sequence to trigger multiple actions in other processes. Syntax 17-2 and 17-4 describe the syntax for declaring named sequences and sequence instances. A sequence instance can be used directly in an event expression, as shown in Syntax 10-9.

When a sequence instance is specified in an event expression, the process executing the event control shall block until the specified sequence reaches its end point. A sequence reaches its end point whenever there is a match for the entire sequence. A process resumes execution following the Observe region in which the end point is detected.

An example of using a sequence as an event control is shown below.

```
sequence abc;
   @(posedge clk) a ##1 b ##1 c;
endsequence

program test;
   initial begin
      @ abc $display( "Saw a-b-c" );
      L1 : ...
   end
```

```
        endprogram
```

In the example above, when the named sequence `abc` reaches its end point, the **initial** block in the program block `test` is unblocked, then displays the string `"Saw a-b-c"`, and continues execution with the statement labeled `L1`. In this case, the end of the sequence acts as the trigger to unblock the event.

A sequence used in an event control is instantiated (as if by an assert property statement); the event control is used to synchronize to the end of the sequence, regardless of its start time. Arguments to these sequences shall be static; automatic variables used as sequence arguments shall result in an error.

10.11 Level-sensitive sequence controls

The execution of procedural code can be delayed until a sequence termination status is true. This is accomplished using the level-sensitive **wait** statement in conjunction with the built-in method that returns the current end status of a named sequence: `triggered`.

The `triggered` sequence method evaluates to true if the given sequence has reached its end point at that particular point in time (in the current time step) and false otherwise. The triggered status of a sequence is set during the Observe region and persists through the remainder of the time step (i.e., until simulation time advances).

For example:

```
    sequence abc;
        @(posedge clk) a ##1 b ##1 c;
    endsequence

    sequence de;
        @(negedge clk) d ##[2:5] e;
    endsequence

    program check;
        initial begin
            wait( abc.triggered || de.triggered );
            if( abc.triggered )
                $display( "abc succeeded" );
            if( de.triggered )
                $display( "de succeeded" );
            L2 : ...
        end
    endprogram
```

In the above example, the **initial** block in program check waits for the end point (success) of either sequence `abc` or sequence `de`. When either condition evaluates to true, the **wait** statement unblocks the process, displays the sequences that caused the process to unblock, and then continues to execute the statement labeled `L2`.

10.12 Procedural assign and deassign removal

SystemVerilog currently supports the procedural **assign** and **deassign** statements. However, these statements might be removed from future versions of the language. See 25.3.

11. Processes

11.1 Introduction

NOTE—Verilog has **always** and **initial** blocks that define static processes.

In an **always** block that is used to model combinational logic, forgetting an **else** leads to an unintended latch. To avoid this mistake, SystemVerilog adds specialized **always_comb** and **always_latch** blocks, which indicate design intent to simulation, synthesis, and formal verification tools. SystemVerilog also adds an **always_ff** block to indicate sequential logic.

In systems modeling, one of the key limitations of Verilog is the inability to create processes dynamically, as happens in an operating system. Verilog has the **fork**...**join** construct, but this still imposes a static limit.

SystemVerilog has both static processes, introduced by **always**, **initial**, or **fork**, and dynamic processes, introduced by built-in **fork**...**join_any** and **fork**...**join_none**.

SystemVerilog creates a thread of execution for each **initial** or **always** block, for each parallel statement in a **fork**...**join** block, and for each dynamic process. Each continuous assignment can also be considered its own thread.

SystemVerilog also introduces dynamic process control constructs that can terminate or wait for processes using their dynamic, parent-child relationship. These are **wait fork** and **disable fork**.

11.2 Combinational logic

SystemVerilog provides a special **always_comb** procedure for modeling combinational logic behavior. For example:

```
always_comb
   a = b & c;

always_comb
   d <= #1ns b & c;
```

The **always_comb** procedure provides functionality that is different from a normal always procedure:

— There is an inferred sensitivity list that includes the expressions defined in 11.2.1.

— The variables written on the left-hand side of assignments shall not be written to by any other process.

— The procedure is automatically triggered once at time zero, after all **initial** and **always** blocks have been started so that the outputs of the procedure are consistent with the inputs.

The SystemVerilog **always_comb** procedure differs from the Verilog **always @*** in the following ways:

— **always_comb** automatically executes once at time zero, whereas **always @*** waits until a change occurs on a signal in the inferred sensitivity list.

— **always_comb** is sensitive to changes within the contents of a function, whereas **always @*** is only sensitive to changes to the arguments of a function.

— Variables on the left-hand side of assignments within an **always_comb** procedure, including variables from the contents of a called function, shall not be written to by any other processes, whereas **always @*** permits multiple processes to write to the same variable.

— Statements in an **always_comb** shall not include those that block, have blocking timing or event controls, or **fork...join** statements.

Software tools can perform additional checks to warn if the behavior within an **always_comb** procedure does not represent combinational logic, such as if latched behavior can be inferred.

11.2.1 Implicit always_comb sensitivities

The implicit sensitivity list of an **always_comb** includes the expansions of the longest static prefix of each variable or select expression that is read within the block or within any function called within the block with the following exceptions:

a) Any expansion of a variable declared within the block or within any function called within the block

b) Any expression that is also written within the block or within any function called within the block

For the definition of the longest static prefix, see 8.11.

Hierarchical function calls and function calls from packages are analyzed as normal functions. References to class objects and method calls of class objects do not add anything to the sensitivity list of an **always_comb**.

11.3 Latched logic

SystemVerilog also provides a special **always_latch** procedure for modeling latched logic behavior. For example:

```
always_latch
    if(ck) q <= d;
```

The **always_latch** procedure determines its sensitivity and executes identically to the **always_comb** procedure. Software tools can perform additional checks to warn if the behavior within an **always_latch** procedure does not represent latched logic.

11.4 Sequential logic

The SystemVerilog **always_ff** procedure can be used to model synthesizable sequential logic behavior. For example:

```
always_ff @(posedge clock iff reset == 0 or posedge reset) begin
    r1 <= reset ? 0 : r2 + 1;
    ...
end
```

The **always_ff** block imposes the restriction that it contains one and only one event control and no blocking timing controls. Variables on the left-hand side of assignments within an **always_ff** procedure, including variables from the contents of a called function, shall not be written to by any other process. Software tools can perform additional checks to warn if the behavior within an **always_ff** procedure does not represent sequential logic.

11.5 Continuous assignments

In Verilog, continuous assignments can only drive nets, and not variables.

SystemVerilog removes this restriction and permits continuous assignments to drive nets and variables of any data type. Nets can be driven by multiple continuous assignments or by a mixture of primitives and continuous assignments. Variables can only be driven by one continuous assignment or one primitive output. It shall be an error for a variable driven by a continuous assignment or primitive output to have an initializer in the declaration or any procedural assignment. See also 6.7.

11.6 fork...join

The **fork**...**join** construct enables the creation of concurrent processes from each of its parallel statements.

The syntax to declare a **fork**...**join** block is as follows:

```
par_block ::=                                                                  // from A.6.3
        fork [ : block_identifier ] { block_item_declaration } { statement_or_null }
        join_keyword [ : block_identifier ]
join_keyword ::= join | join_any | join_none
```

Syntax 11-1—Fork...join block syntax (excerpt from Annex A)

One or more statements can be specified; each statement shall execute as a concurrent process.

A Verilog **fork**...**join** block always causes the process executing the fork statement to block until the termination of all forked processes. With the addition of the **join_any** and **join_none** keywords, System-Verilog provides three choices for specifying when the parent (forking) process resumes execution (see Table 11-1).

Table 11-1—Fork...join control options

Option	Description
join	The parent process blocks until all the processes spawned by this fork complete.
join_any	The parent process blocks until any one of the processes spawned by this fork completes.
join_none	The parent process continues to execute concurrently with all the processes spawned by the fork. The spawned processes do not start executing until the parent thread executes a blocking statement.

When defining a **fork**...**join** block, encapsulating the entire fork within a **begin**...**end** block causes the entire block to execute as a single process, with each statement executing sequentially.

```
    fork
       begin
          statement1;     // one process with 2 statements
          statement2;
       end
    join
```

In the following example, two processes are forked. The first one waits for 20 ns and the second waits for the named event eventA to be triggered. Because the **join** keyword is specified, the parent process shall block until the two processes complete, i.e., until 20 ns have elapsed and eventA has been triggered.

```
    fork
       begin
          $display( "First Block\n" );
          # 20ns;
       end
       begin
          $display( "Second Block\n" );
```

```
        @eventA;
      end
   join
```

A `return` statement within the context of a `fork`...`join` statement is illegal and shall result in a compilation error. For example:

```
   task wait_20;
      fork
         # 20;
         return ;    // Illegal: cannot return; task lives in another process
      join_none
   endtask
```

Variables declared in the *block_item_declaration* of a `fork`...`join`, `join_any`, or `join_none` block shall be initialized to their initialization value expression whenever execution enters their scope and before any processes are spawned. Within a `fork...join_any` or `fork`...`join_none` block, it shall be illegal to refer to formal arguments passed by reference other than in the initialization value expressions of variables declared in a *block_item_declaration* of the fork. These variables are useful in processes spawned by looping constructs to store unique, per-iteration data. For example:

```
   initial
      for( int j = 1; j <= 3; ++j )
         fork
            automatic int k = j; // local copy, k, for each value of j
            #k $write( "%0d", k );
            begin
               automatic int m = j; // the value of m is undetermined
               ...
            end
         join_none
```

The example above generates the output 123.

11.7 Process execution threads

SystemVerilog creates a thread of execution for the following:

— Each `initial` block

— Each `always` block

— Each parallel statement in a `fork`...`join` (or `join_any` or `join_none`) statement group

— Each dynamic process

Each continuous assignment can also be considered its own thread.

11.8 Process control

SystemVerilog provides constructs that allow one process to terminate or wait for the completion of other processes. The `wait fork` construct waits for the completion of processes. The `disable fork` construct stops the execution of processes.

11.8.1 Wait fork

The **wait fork** statement is used to ensure that all child processes (processes created by the calling process) have completed their execution.

The syntax for **wait fork** is as follows:

```
wait fork ; // from A.6.5
```

Specifying **wait fork** causes the calling process to block until all its subprocesses have completed.

Verilog terminates a simulation run when there is no further activity of any kind. SystemVerilog adds the ability to automatically terminate the simulation when all its program blocks finish executing (i.e, they reach the end of their execute block), regardless of the status of any child processes (see 16.6). The **wait fork** statement allows a program block to wait for the completion of all its concurrent threads before exiting.

In the following example, in the task do_test, the first two processes are spawned, and the task blocks until one of the two processes completes (either exec1 or exec2). Next, two more processes are spawned in the background. The **wait fork** statement shall ensure that the task do_test waits for all four spawned processes to complete before returning to its caller.

```
task do_test;
   fork
      exec1();
      exec2();
   join_any
   fork
      exec3();
      exec4();
   join_none
   wait fork;        // block until exec1 ... exec4 complete
endtask
```

11.8.2 Disable fork

The **disable fork** statement terminates all active descendants (subprocesses) of the calling process.

The syntax for **disable fork** is as follows:

```
disable fork ; // from A.6.5
```

The **disable fork** statement terminates all descendants of the calling process as well as the descendants of the process's descendants. In other words, if any of the child processes have descendants of their own, the **disable fork** statement shall terminate them as well.

In the example below, the task get_first spawns three versions of a task that wait for a particular device (1, 7, or 13). The task wait_device waits for a particular device to become ready and then returns the device's address. When the first device becomes available, the get_first task shall resume execution and proceed to kill the outstanding wait_device processes.

```
task get_first( output int adr );
   fork
      wait_device( 1, adr );
      wait_device( 7, adr );
      wait_device( 13, adr );
   join_any
```

```
        disable fork;
    endtask
```

Verilog supports the **disable** construct, which terminates a process when applied to the named block being executed by the process. The **disable fork** statement differs from **disable** in that **disable fork** considers the dynamic parent-child relationship of the processes, whereas **disable** uses the static, syntactical information of the disabled block. Thus, **disable** shall end all processes executing a particular block, whether the processes were forked by the calling thread or not, while **disable fork** shall end only the processes that were spawned by the calling thread.

11.9 Fine-grain process control

A process is a built-in class that allows one process to access and control another process once it has started. Users can declare variables of type process and safely pass them through tasks or incorporate them into other objects. The prototype for the process class is as follows:

```
class process;
    enum state { FINISHED, RUNNING, WAITING, SUSPENDED, KILLED };

    static function process self();
    function state status();
    function void kill();
    task await();
    function void suspend();
    task resume();
endclass
```

Objects of type process are created internally when processes are spawned. Users cannot create objects of type process; attempts to call **new** shall not create a new process and shall instead result in an error. The process class cannot be extended. Attempts to extend it shall result in a compilation error. Objects of type process are unique; they become available for reuse once the underlying process terminates and all references to the object are discarded.

The **self()** function returns a handle to the current process, that is, a handle to the process making the call.

The **status()** function returns the process status, as defined by the state enumeration:
— **FINISHED** means the process terminated normally.
— **RUNNING** means the process is currently running (not in a blocking statement).
— **WAITING** means the process is waiting in a blocking statement.
— **SUSPENDED** means the process is stopped awaiting a resume.
— **KILLED** means the process was forcibly killed (via kill or disable).

The **kill()** task terminates the given process and all its subprocesses, that is, processes spawned using **fork** statements by the process being killed. If the process to be terminated is not blocked waiting on some other condition, such as an event, **wait** expression, or a delay, then the process shall be terminated at some unspecified time in the current time step.

The **await()** task allows one process to wait for the completion of another process. It shall be an error to call this task on the current process, i.e., a process cannot wait for its own completion.

The **suspend()** task allows a process to suspend either its own execution or that of another process. If the process to be suspended is not blocked waiting on some other condition, such as an event, **wait** expression,

or a delay, then the process shall be suspended at some unspecified time in the current time step. Calling this method more than once, on the same (suspended) process, has no effect.

The `resume()` task restarts a previously suspended process. Calling resume on a process that was suspended while blocked on another condition shall resensitize the process to the event expression or to wait for the wait condition to become true or for the delay to expire. If the wait condition is now true or the original delay has transpired, the process is scheduled onto the Active or Reactive region to continue its execution in the current time step. Calling resume on a process that suspends itself causes the process to continue to execute at the statement following the call to suspend.

The example below starts an arbitrary number of processes, as specified by the task argument *N*. Next, the task waits for the last process to start executing and then waits for the first process to terminate. At that point, the parent process forcibly terminates all forked processes that have not completed yet.

```
task do_n_way( int N );
   process job[1:N];

   for ( int j = 1; j <= N; j++ )
      fork
         automatic int k = j;
         begin job[j] = process::self(); ... ; end
      join_none

   for( int j = 1; j <= N; j++ )    // wait for all processes to start
      wait( job[j] != null );

   job[1].await();                   // wait for first process to finish

   for ( int k = 1; k <= N; k++ ) begin
      if ( job[k].status != process::FINISHED )
         job[k].kill();
   end
endtask
```

12. Tasks and functions

12.1 Introduction

NOTE—Verilog has static and automatic tasks and functions. Static tasks and functions share the same storage space for all calls to the tasks or function within a module instance. Automatic tasks and function allocate unique, stacked storage for each instance.

SystemVerilog adds the ability to declare automatic variables within static tasks and functions and to declare static variables within automatic tasks and functions.

SystemVerilog also adds the following:

— More capabilities for declaring task and function ports
— Function output and inout ports
— Void functions
— Multiple statements in a task or function without requiring a **begin**...**end** or **fork**...**join** block
— Returning from a task or function before reaching the end of the task or function
— Passing arguments by reference instead of by value
— Binding argument values by name instead of by position
— Default argument values
— Importing and exporting functions through the DPI

12.2 Tasks

```
task_declaration ::= task [ lifetime ] task_body_declaration                          // from A.2.7

task_body_declaration ::=
        [ interface_identifier . | class_scope ] task_identifier ;
        { tf_item_declaration }
        { statement_or_null }
        endtask [ : task_identifier ]
    | [ interface_identifier . | class_scope ] task_identifier ( [ tf_port_list ] ) ;
        { block_item_declaration }
        { statement_or_null }
        endtask [ : task_identifier ]

tf_item_declaration ::=
        block_item_declaration
    | tf_port_declaration

tf_port_list ::=
        tf_port_item { , tf_port_item }

tf_port_item[34] ::=
        { attribute_instance }
            [ tf_port_direction ] [ var ] data_type_or_implicit
            [ port_identifier { variable_dimension } [ = expression ] ]

tf_port_direction ::= port_direction | const ref

tf_port_declaration ::=
        { attribute_instance } tf_port_direction [ var ] data_type_or_implicit list_of_tf_variable_identifiers ;
```

Syntax 12-1—Task syntax (excerpt from Annex A)

lifetime ::= **static** | **automatic**   *// from A.2.1*

signing ::= **signed** | **unsigned**   *// from A.2.2.1*

data_type_or_implicit ::=
      data_type
   | [signing] { packed_dimension }

34. In a tf_port_item, it shall be illegal to omit the explicit port_identifier except within a function_prototype or task_prototype.

Syntax 12-1—Task syntax (excerpt from Annex A) (continued)

A Verilog task declaration has the formal arguments either in parentheses (like ANSI C) or in declarations and directions.

```
task mytask1 (output int x, input logic y);
   ...
endtask

task mytask2;
   output x;
   input y;
   int x;
   logic y;
   ...
endtask
```

Each formal argument has one of the following directions:

```
input        // copy value in at beginning
output       // copy value out at end
inout        // copy in at beginning and out at end
ref          // pass reference (see 12.4.2)
```

With SystemVerilog, there is a default direction of **input** if no direction has been specified. Once a direction is given, subsequent formals default to the same direction. In the following example, the formal arguments a and b default to inputs, and u and v are both outputs.

```
task mytask3(a, b, output logic [15:0] u, v);
   ...
endtask
```

Each formal argument also has a data type that can be explicitly declared or can inherit a default type. The task argument default type in SystemVerilog is **logic**.

SystemVerilog allows an array to be specified as a formal argument to a task. For example:

```
// the resultant declaration of b is input [3:0][7:0] b[3:0]
task mytask4(input [3:0][7:0] a, b[3:0], output [3:0][7:0] y[1:0]);
   ...
endtask
```

Verilog allows tasks to be declared as **automatic** so that all formal arguments and local variables are stored on the stack. SystemVerilog extends this capability by allowing specific formal arguments and local variables to be declared as **automatic** within a static task or by declaring specific formal arguments and local variables as **static** within an automatic task.

With SystemVerilog, multiple statements can be written between the task declaration and **endtask**; therefore, the **begin** .... **end** can be omitted. If **begin** .... **end** is omitted, statements are executed sequentially, the same as if they were enclosed in a **begin** .... **end** group. It shall also be legal to have no statements at all.

In Verilog, a task exits when the **endtask** is reached. With SystemVerilog, the **return** statement can be used to exit the task before the **endtask** keyword.

12.3 Functions

---

function_data_type ::= data_type | **void**   *// from A.2.6*

function_data_type_or_implicit ::=
        function_data_type
    | [signing] { packed_dimension }

function_declaration ::= **function** [lifetime] function_body_declaration

function_body_declaration ::=
        function_data_type_or_implicit
            [interface_identifier **.** | class_scope] function_identifier **;**
        { tf_item_declaration }
        { function_statement_or_null }
        endfunction [**:** function_identifier]
    | function_data_type_or_implicit
            [interface_identifier **.** | class_scope] function_identifier **(** [tf_port_list] **) ;**
        { block_item_declaration }
        { function_statement_or_null }
        endfunction [**:** function_identifier]

lifetime ::= **static** | **automatic**   *// from A.2.1.3*

signing ::= **signed** | **unsigned**   *// from A.2.2.1*

Syntax 12-2—Function syntax (excerpt from Annex A)

A Verilog function declaration has the formal arguments either in parentheses (like ANSI C) or in declarations and directions:

```
function logic [15:0] myfunc1(int x, int y);
   ...
endfunction

function logic [15:0] myfunc2;
   input int x;
   input int y;
   ...
endfunction
```

SystemVerilog extends Verilog functions to allow the same formal arguments as tasks. Function argument directions are as follows:

```
input        // copy value in at beginning
output       // copy value out at end
inout        // copy in at beginning and out at end
ref          // pass reference (see 12.4.2)
```

Function declarations default to the formal direction **input** if no direction has been specified. Once a direction is given, subsequent formals default to the same direction. In the following example, the formal arguments a and b default to inputs, and u and v are both outputs:

```
function logic [15:0] myfunc3(int a, int b, output logic [15:0] u, v);
    ...
endfunction
```

Each formal argument has a data type that can be explicitly declared or can inherit a default type. The default type in SystemVerilog is **logic**, which is compatible with Verilog. SystemVerilog allows an array to be specified as a formal argument to a function, for example:

```
function [3:0][7:0] myfunc4(input [3:0][7:0] a, b[3:0]);
    ...
endfunction
```

It shall be illegal to call a function with **output**, **inout**, or **ref** arguments in an event expression, in an expression within a procedural continuous assignment, or in an expression that is not within a procedural statement. However, a **const ref** function argument shall be legal in this context (see 12.4.2).

SystemVerilog allows multiple statements to be written between the function header and **endfunction**; therefore, the **begin**...**end** can be omitted. If the **begin**...**end** is omitted, statements are executed sequentially, as if they were enclosed in a **begin**...**end** group. It is also legal to have no statements at all, in which case the function returns the current value of the implicit variable that has the same name as the function.

12.3.1 Return values and void functions

In Verilog, functions must return values. The return value is specified by assigning a value to the name of the function.

```
function [15:0] myfunc1 (input [7:0] x,y);
    myfunc1 = x * y - 1; //return value is assigned to function name
endfunction
```

SystemVerilog allows functions to be declared as type **void**, which do not have a return value. For nonvoid functions, a value can be returned by assigning the function name to a value, as in Verilog, or by using **return** with a value. The **return** statement shall override any value assigned to the function name. When the return statement is used, nonvoid functions must specify an expression with the return.

```
function [15:0] myfunc2 (input [7:0] x,y);
    return x * y - 1; //return value is specified using return statement
endfunction
```

In SystemVerilog, a function return can be a structure or union. In this case, a hierarchical name used inside the function and beginning with the function name is interpreted as a member of the return value. If the function name is used outside the function, the name indicates the scope of the whole function. If the function name is used within a hierarchical name, it also indicates the scope of the whole function.

Function calls are expressions unless of type **void**, which are statements:

```
a = b + myfunc1(c, d); //call myfunc1 (defined above) as an expression

myprint(a); //call myprint (defined below) as a statement

function void myprint (int a);
    ...
```

```
        endfunction
```

12.3.2 Discarding function return values

In Verilog, values returned by functions must be assigned or used in an expression. Calling a function as if it has no return value can result in a warning message. SystemVerilog allows using the **void** data type to discard a function's return value. Discarding is done by casting the function to the **void** type:

```
        void'(some_function());
```

12.3.3 Constant function calls

In SystemVerilog, a constant function call is a function invocation of a constant function local to the calling module or from a package or $unit where the arguments to the function are all constant expressions. Verilog has a number of restrictions that constant functions shall meet. The Verilog restrictions for constant functions are modified in SystemVerilog as follows:

— A constant function may call any system function that may be called in a constant expression. This includes $bits and the array query functions.

— A constant function may reference parameters defined in packages or $unit.

SystemVerilog adds the following additional restrictions on constant functions:

— A constant function shall not have output, inout, or ref arguments.

— A void function shall not be a constant function.

— An import "DPI" function (see 12.5) shall not be a constant function.

— A constant function may have default argument values (see 12.4.3), but any such default argument value shall be a constant expression.

12.4 Task and function argument passing

SystemVerilog provides two means for passing arguments to functions and tasks: by value and by reference. Arguments can also be bound by name as well as by position. Task and function arguments can also be given default values, allowing the call to the task or function to not pass arguments.

12.4.1 Pass by value

Pass by value is the default mechanism for passing arguments to subroutines. This argument passing mechanism works by copying each argument into the subroutine area. If the subroutine is automatic, then the subroutine retains a local copy of the arguments in its stack. If the arguments are changed within the subroutine, the changes are not visible outside the subroutine. When the arguments are large, it can be undesirable to copy the arguments. Also, programs sometimes need to share a common piece of data that is not declared global.

For example, calling the function below copies 1000 bytes each time the call is made.

```
    function int crc( byte packet [1000:1] );
      for( int j= 1; j <= 1000; j++ ) begin
        crc ^= packet[j];
      end
    endfunction
```

12.4.2 Pass by reference

Arguments passed by reference are not copied into the subroutine area, rather, a reference to the original argument is passed to the subroutine. The subroutine can then access the argument data via the reference. Arguments passed by reference must be matched with equivalent data types (see 6.9.2). No casting shall be permitted. To indicate argument passing by reference, the argument declaration is preceded by the **ref** keyword. It shall be illegal to use argument passing by reference for subroutines with a lifetime of **static**. The general syntax is as follows:

```
subroutine( ref type argument );
```

For example, the example above can be written as follows:

```
function int crc( ref byte packet [1000:1] );
    for( int j= 1; j <= 1000; j++ ) begin
        crc ^= packet[j];
    end
endfunction
```

As shown in the preceding example, no change other than addition of the **ref** keyword is needed. The compiler knows that packet is now addressed via a reference, but users do not need to make these references explicit either in the callee or at the point of the call. In other words, the call to either version of the crc function remains the same:

```
byte packet1[1000:1];
int k = crc( packet1 ); // pass by value or by reference: call is the same
```

When the argument is passed by reference, both the caller and the subroutine share the same representation of the argument; therefore, any changes made to the argument, within either the caller or the subroutine, shall be visible to each other. The semantics of assignments to variables passed by reference is that changes are seen outside the subroutine immediately (before the subroutine returns). Only variables, not nets, can be passed by reference.

Because a variable passed by reference may be an automatic variable, a **ref** argument shall not be used in any context forbidden for automatic variables.

Elements of dynamic arrays, queues, and associative arrays that are passed by reference may get removed from the array or the array may get resized before the called function or task completes. The specific array element passed by reference shall continue to exist within the scope of the called tasks or functions until they complete. Changes made to the values of array elements by the called task or function shall not be visible outside the scope of those tasks or functions if those array elements were removed from the array before the changes were made. These references shall be called *outdated references*.

The following operations on a variable-size array shall cause existing references to elements of that array to become outdated references:

— A dynamic array is resized with an implicit or explicit **new**[].
— A dynamic array is deleted with the delete() method.
— The element of an associative array being referenced is deleted with the delete() method.
— A queue is assigned with an array aggregate expression that does not explicitly contain the element being referenced.
— The element of a queue being referenced is deleted by a queue method.

Passing an argument by reference is a unique argument-passing qualifier, different from **input**, **output**, or **inout**. Combining **ref** with any other directional qualifier shall be illegal. For example, the following declaration results in a compiler error:

```
task incr( ref input int a ); // incorrect: ref cannot be qualified
```

A **ref** argument is similar to an **inout** argument except that an **inout** argument is copied twice: once from the actual into the argument when the subroutine is called and once from the argument into the actual when the subroutine returns. Passing object handles is no exception and has similar semantics when passed as **ref** or **inout** arguments. Thus, a **ref** of an object handle allows changes to the object handle (for example, assigning a new object) in addition to modification of the contents of the object.

To protect arguments passed by reference from being modified by a subroutine, the **const** qualifier can be used together with **ref** to indicate that the argument, although passed by reference, is a read-only variable.

```
task show ( const ref byte [] data );
   for ( int j = 0; j < data.size ; j++ )
       $display( data[j] ); // data can be read but not written
endtask
```

When the formal argument is declared as a **const ref**, the subroutine cannot alter the variable, and an attempt to do so shall generate a compiler error.

12.4.3 Default argument values

To handle common cases or allow for unused arguments, SystemVerilog allows a subroutine declaration to specify a default value for each singular argument.

The syntax to declare a default argument in a subroutine is as follows:

```
subroutine( [ direction ] [ type ] argument = default_value );
```

The optional direction can be **input**, **inout**, or **ref** (output ports cannot specify defaults).

The default_value is an expression. The expression is evaluated in the scope containing the subroutine declaration each time a call using the default is made. If the default_value is not used, the expression is not evaluated. The use of default values shall only be allowed with the ANSI style declarations.

When the subroutine is called, arguments with default values can be omitted from the call, and the compiler shall insert their corresponding values. Unspecified (or empty) arguments can be used as placeholders for default arguments, allowing the use of nonconsecutive default arguments. If an unspecified argument is used for an argument that does not have a default value, a compiler error shall be issued.

```
task read(int j = 0, int k, int data = 1 );
...
endtask;
```

This example declares a task read() with two default arguments, j and data. The task can then be called using various default arguments:

```
read( , 5 );          // is equivalent to  read( 0, 5, 1 );
read( 2, 5 );         // is equivalent to  read( 2, 5, 1 );
read( , 5, );         // is equivalent to  read( 0, 5, 1 );
read( , 5, 7 );       // is equivalent to  read( 0, 5, 7 );
read( 1, 5, 2 );      // is equivalent to  read( 1, 5, 2 );
```

```
        read( );                // error; k has no default value
```

12.4.4 Argument binding by name

SystemVerilog allows arguments to tasks and functions to be bound by name as well as by position. This allows specifying nonconsecutive default arguments and easily specifying the argument to be passed at the call. For example:

```
        function int fun( int j = 1, string s = "no" );
            ...
        endfunction
```

The fun function can be called as follows:

```
    fun( .j(2), .s("yes") );    // fun( 2, "yes" );
    fun( .s("yes") );           // fun( 1, "yes" );
    fun( , "yes" );             // fun( 1, "yes" );
    fun( .j(2) );               // fun( 2, "no" );
    fun( .s("yes"), .j(2) );    // fun( 2, "yes" );
    fun( .s(), .j() );          // fun( 1, "no" );
    fun( 2 );                   // fun( 2, "no" );
    fun( );                     // fun( 1, "no" );
```

If the arguments have default values, they are treated like parameters to module instances. If the arguments do not have a default, then they must be given, or the compiler shall issue an error.

If both positional and named arguments are specified in a single subroutine call, then all the positional arguments must come before the named arguments. Then, using the same example as above:

```
    fun( .s("yes"), 2 );      // illegal
    fun( 2, .s("yes") );      // OK
```

12.4.5 Optional argument list

When a void function or class function method specifies no arguments, the empty parenthesis, (), following the subroutine name shall be optional. This is also true for tasks, void functions, and class methods that require arguments, when all arguments have defaults specified. It shall be illegal to omit the parenthesis in a directly recursive nonvoid function method call that is not hierarchically qualified.

12.5 Import and export functions

The syntax for the import and export of functions is as follows:

---

dpi_import_export ::=   // from A.2.6
    import dpi_spec_string [dpi_function_import_property] [c_identifier **=**] dpi_function_proto **;**
    | **import** dpi_spec_string [dpi_task_import_property] [c_identifier **=**] dpi_task_proto **;**
    | **export** dpi_spec_string [c_identifier **=**] **function** function_identifier **;**
    | **export** dpi_spec_string [c_identifier =] **task** task_identifier **;**

dpi_spec_string ::= **"DPI-C"** | **"DPI"**

dpi_import_property ::= **context** | **pure**

dpi_function_proto[8,9] ::= function_prototype

Syntax 12-3—Import and export syntax (excerpt from Annex A)

In both **import** and **export**, *c_identifier* is the name of the foreign function (import/export), and *function_identifier* is the SystemVerilog name for the same function. If *c_identifier* is not explicitly given, it shall be the same as the SystemVerilog function *function_identifier*. An error shall be generated if, and only if, the *c_identifier* has characters that are not valid in a C function identifier.

Several SystemVerilog functions can be mapped to the same foreign function by supplying the same *c_identifier* for several *fnames*. The corresponding SystemVerilog functions must have identical argument types, as defined in the next paragraph.

For any given *c_identifier*, all declarations, regardless of scope, must have exactly the same function signature. The function signature includes the return type and the number, order, direction, and types of each and every argument. Each type includes dimensions and bounds of any arrays/array dimensions. For **import** declarations, arguments can be open arrays. Open arrays are defined in 26.4.6.1. The signature also includes the **pure**/**context** qualifiers that can be associated with an import definition.

Only one **import** or **export** declaration of a given *function_identifier* shall be permitted in any given scope. More specifically, for an **import**, the import must be the sole declaration of *function_identifier* in the given scope. For an **export**, the function must be declared in the scope where the export occurs, and there must be only one export of that *function_identifier* in that scope.

For exported functions, the exported function must be declared in the same scope that contains the **export** "DPI" declaration. Only SystemVerilog functions can be exported (specifically, this excludes exporting a class method).

All **import** "DPI" functions declared this way can be invoked by hierarchical reference the same as any normal SystemVerilog function. Declaring a SystemVerilog function to be exported does not change the semantics or behavior of this function from the SystemVerilog perspective (i.e., there is no effect in System-Verilog usage other than making this exported function also accessible to C callers).

Only nonvoid functions with no **output** or **inout** arguments can be specified as **pure**. Functions specified as pure in their corresponding SystemVerilog external declarations shall have no side effects; their results need to depend solely on the values of their input arguments. Calls to such functions can be removed by SystemVerilog compiler optimizations or replaced with the values previously computed for the same values of the input arguments.

Specifically, a pure function is assumed to not directly or indirectly (i.e., by calling other functions) perform the following:
— Perform any file operations
— Read or write anything in the broadest possible meaning, including input/output, environment variables, objects from the operating system, or from the program or other processes, shared memory, sockets, etc.
— Access any persistent data, like global or static variables

If a pure function does not obey the above restrictions, SystemVerilog compiler optimizations can lead to unexpected behavior, due to eliminated calls or incorrect results being used.

An unqualified imported function can have side effects, but cannot read or modify any SystemVerilog signals other than those provided through its arguments. Unqualified imports shall not be permitted to invoke exported SystemVerilog functions.

Imported functions with the **context** qualifier can invoke exported SystemVerilog functions and can read or write to SystemVerilog signals other than those passed through their arguments, either through the use of other interfaces or as a side effect of invoking exported SystemVerilog functions. Context functions shall

always implicitly be supplied a scope representing the fully qualified instance name within which the **import** declaration was present (i.e., an import function always runs in the instance in which the **import** declaration occurred). This is the same semantics as SystemVerilog functions, which also run in the scope they are defined, rather than in the scope of the caller.

Import context functions can have side effects and can use other SystemVerilog interfaces (including but not limited to VPI). However, declaring an import context function does not automatically make any other simulator interface available. For VPI access (or any other interface access) to be possible, the appropriate implementation-defined mechanism must still be used to enable these interface(s). Also, SystemVerilog DPI calls do not automatically create or provide any handles or any special environment that might be needed by the other interfaces. It shall be the user's responsibility to create, manage, or otherwise manipulate the required handles/environment(s) needed by the other interfaces. The `svGetScopeName()` and related functions exist to provide a name-based linkage from DPI to other interfaces. Exported functions can only be invoked if the current DPI context refers to an instance in which the named function is defined.

To access functions defined in any other scope, the foreign code shall have to change DPI context appropriately. Attempting to invoke an exported SystemVerilog function from a scope in which it is not directly visible shall result in a run-time error. How such errors are handled shall be implementation dependent. If an imported function needs to invoke an exported function that is not visible from the current scope, it needs to change, via `svSetScope`, the current scope to a scope that does have visibility to the exported function. This is conceptually equivalent to making a hierarchically qualified function call in SystemVerilog. The current SystemVerilog context shall be preserved across a call to an exported function, even if current context has been modified by an application. For noncontext imports, the context is not defined, and attempting to use any functionality depending on context from noncontext imports can lead to unpredictable behavior.

13. Random constraints

13.1 Introduction

NOTE—Constraint-driven test generation allows users to automatically generate tests for functional verification. Random testing can be more effective than a traditional, directed testing approach. By specifying constraints, one can easily create tests that can find hard-to-reach corner cases. SystemVerilog allows users to specify constraints in a compact, declarative way. The constraints are then processed by a solver that generates random values that meet the constraints.

The random constraints are typically specified on top of an object-oriented data abstraction that models the data to be randomized as objects that contain random variables and user-defined constraints. The constraints determine the legal values that can be assigned to the random variables. Objects are ideal for representing complex aggregate data types and protocols such as Ethernet packets.

Subclause 13.2 provides an overview of object-based randomization and constraint programming. The rest of this clause provides detailed information on random variables, constraint blocks, and the mechanisms used to manipulate them.

13.2 Overview

This subclause introduces the basic concepts and uses for generating random stimulus within objects. SystemVerilog uses an object-oriented method for assigning random values to the member variables of an object, subject to user-defined constraints. For example:

```
class Bus;
   rand bit[15:0] addr;
   rand bit[31:0] data;

   constraint word_align {addr[1:0] == 2’b0;}
endclass
```

The `Bus` class models a simplified bus with two random variables: `addr` and `data`, representing the address and data values on a bus. The `word_align` constraint declares that the random values for `addr` must be such that `addr` is word-aligned (the low-order 2 bits are 0).

The `randomize()` method is called to generate new random values for a bus object:

```
Bus bus = new;

repeat (50) begin
   if ( bus.randomize() == 1 )
      $display ("addr = %16h data = %h\n", bus.addr, bus.data);
   else
      $display ("Randomization failed.\n");
end
```

Calling `randomize()` causes new values to be selected for all of the random variables in an object so that all of the constraints are true (satisfied). In the program test above, a `bus` object is created and then randomized 50 times. The result of each randomization is checked for success. If the randomization succeeds, the new random values for `addr` and `data` are printed; if the randomization fails, an error message is printed. In this example, only the `addr` value is constrained, while the `data` value is unconstrained. Unconstrained variables are assigned any value in their declared range.

Constraint programming is a powerful method that lets users build generic, reusable objects that can later be extended or constrained to perform specific functions. The approach differs from both traditional procedural and object-oriented programming, as illustrated in this example that extends the `Bus` class:

```
typedef enum {low, mid, high} AddrType;
```

```
class MyBus extends Bus;
    rand AddrType atype;
    constraint addr_range
    {
        (atype == low ) -> addr inside { [0 : 15] };
        (atype == mid ) -> addr inside { [16 : 127]};
        (atype == high) -> addr inside {[128 : 255]};
    }
endclass
```

The MyBus class inherits all of the random variables and constraints of the Bus class and adds a random variable called atype that is used to control the address range using another constraint. The addr_range constraint uses implication to select one of three range constraints depending on the random value of atype. When a MyBus object is randomized, values for addr, data, and atype are computed so that all of the constraints are satisfied. Using inheritance to build layered constraint systems enables the development of general-purpose models that can be constrained to perform application-specific functions.

Objects can be further constrained using the randomize() with construct, which declares additional constraints in line with the call to randomize():

```
task exercise_bus (MyBus bus);
    int res;

    // EXAMPLE 1: restrict to low addresses
    res = bus.randomize() with {atype == low;};

    // EXAMPLE 2: restrict to address between 10 and 20
    res = bus.randomize() with {10 <= addr && addr <= 20;};

    // EXAMPLE 3: restrict data values to powers-of-two
    res = bus.randomize() with {data & (data - 1) == 0;};
endtask
```

This example illustrates several important properties of constraints:
— Constraints can be any SystemVerilog expression with variables and constants of integral type (e.g., **bit**, **reg**, **logic**, **integer**, **enum**, **packed struct**).
— The constraint solver must be able to handle a wide spectrum of equations, such as algebraic factoring, complex boolean expressions, and mixed integer and bit expressions. In the example above, the power-of-two constraint was expressed arithmetically. It could have also been defined with expressions using a shift operator. For example, 1 << n, where n is a 5-bit random variable.
— If a solution exists, the constraint solver must find it. The solver can fail only when the problem is over-constrained and there is no combination of random values that satisfy the constraints.
— Constraints interact bidirectionally. In this example, the value chosen for addr depends on atype and how it is constrained, and the value chosen for atype depends on addr and how it is constrained. All expression operators are treated bidirectionally, including the implication operator (->).
— Constraints support only 2-state values. The 4-state values (X or Z) or 4-state operators (e.g., ===, !==) are illegal and shall result in an error.

Sometimes it is desirable to disable constraints on random variables. For example, to deliberately generate an illegal address (nonword-aligned):

```
task exercise_illegal(MyBus bus, int cycles);
    int res;
```

```
    // Disable word alignment constraint.
    bus.word_align.constraint_mode(0);

    repeat (cycles) begin

    // CASE 1: restrict to small addresses.
    res = bus.randomize() with {addr[0] || addr[1];};
        ...
    end

    // Reenable word alignment constraint
    bus.word_align.constraint_mode(1);
endtask
```

The `constraint_mode()` method can be used to enable or disable any named constraint block in an object. In this example, the word-alignment constraint is disabled, and the object is then randomized with additional constraints forcing the low-order address bits to be nonzero (and thus unaligned).

The ability to enable or disable constraints allows users to design constraint hierarchies. In these hierarchies, the lowest level constraints can represent physical limits grouped by common properties into named constraint blocks, which can be independently enabled or disabled.

Similarly, the `rand_mode()` method can be used to enable or disable any random variable. When a random variable is disabled, it behaves in exactly the same way as other nonrandom variables.

Occasionally, it is desirable to perform operations immediately before or after randomization. That is accomplished via two built-in methods, `pre_randomize()` and `post_randomize()`, which are automatically called before and after randomization. These methods can be overridden with the desired functionality:

```
class XYPair;
    rand integer x, y;
endclass

class MyXYPair extends XYPair
    function void pre_randomize();
        super.pre_randomize();
        $display("Before randomize x=%0d, y=%0d", x, y);
    endfunction

    function void post_randomize();
        super.post_randomize();
        $display("After randomize x=%0d, y=%0d", x, y);
    endfunction
endclass
```

By default, `pre_randomize()` and `post_randomize()` call their overridden parent class methods. When `pre_randomize()` or `post_randomize()` are overridden, care must be taken to invoke the parent class's methods, unless the class is a base class (has no parent class). Otherwise, the base class methods shall not be called.

The random stimulus generation capabilities and the object-oriented constraint-based verification methodology enable users to quickly develop tests that cover complex functionality and better assure design correctness.

13.3 Random variables

Class variables can be declared random using the **rand** and **randc** type-modifier keywords.

The syntax to declare a random variable in a class is as follows:

```
class_property ::=                                              // from A.1.8
        { property_qualifier } data_declaration
property_qualifier7 ::=
        random_qualifier
      | class_item_qualifier
random_qualifier7 ::=
        rand
      | randc
```

Syntax 13-1—Random variable declaration syntax (excerpt from Annex A)

— The solver can randomize singular variables of any integral type.
— Arrays can be declared **rand** or **randc**, in which case all of their member elements are treated as **rand** or **randc**.
— Individual array elements can be constrained, in which case the index expression must be a literal constant.
— If the array elements are object handles, all of the array elements must be non-null.
— Dynamic and associative arrays can be declared **rand** or **randc**. All of the elements in the array are randomized, overwriting any previous data.
— The size of a dynamic array declared as **rand** or **randc** can also be constrained. In that case, the array shall be resized according to the size constraint, and then all the array elements shall be randomized. The array size constraint is declared using the `size` method. For example,

```
rand bit [7:0] len;
rand integer data[];
constraint db { data.size == len; }
```

The variable `len` is declared to be 8 bits wide. The randomizer computes a random value for the `len` variable in the 8-bit range of 0 to 255 and then randomizes the first `len` elements of the data array.

If a dynamic array's size is not constrained, then `randomize()` randomizes all the elements in the array.

— An object handle can be declared **rand**, in which case all of that object's variables and constraints are solved concurrently with the variables and constraints of the object that contains the handle. Objects cannot be declared **randc**.
— An unpacked structure can be declared **rand**, in which case all of that structure's random members are solved concurrently using one of the rules listed in this subclause. Unpacked structures shall not be declared **randc**. A member of a unpacked structure can be made random by having a **rand** or **randc** modifier in the declaration of its type. Members of unpacked structures containing a union as well as members of packed structures shall not be allowed to have a random modifier.

For example:

```
class packet;
typedef struct {
   randc int addr = 1 + constant;
```

```
        int crc;
        rand byte data [] = {1,2,3,4};
    } header;
    rand header h1;
    endclass
    packet p1=new;
```

13.3.1 Rand modifier

Variables declared with the **rand** keyword are standard random variables. Their values are uniformly distributed over their range. For example:

```
    rand bit [7:0] y;
```

This is an 8-bit unsigned integer with a range of 0 to 255. If unconstrained, this variable shall be assigned any value in the range of 0 to 255 with equal probability. In this example, the probability of the same value repeating on successive calls to randomize is 1/256.

13.3.2 Randc modifier

Variables declared with the **randc** keyword are random-cyclic variables that cycle through all the values in a random permutation of their declared range. Random-cyclic variables can only be of type **bit** or enumerated types and can be limited to a maximum size.

To understand **randc**, consider a 2-bit random variable y:

```
    randc bit [1:0] y;
```

The variable y can take on the values 0, 1, 2, and 3 (range of 0 to 3). Randomize computes an initial random permutation of the range values of y and then returns those values in order on successive calls. After it returns the last element of a permutation, it repeats the process by computing a new random permutation.

The basic idea is that **randc** randomly iterates over all the values in the range and that no value is repeated within an iteration. When the iteration finishes, a new iteration automatically starts (see Figure 13-1).

initial permutation: 0 → 3 → 2 → 1

next permutation: → 2 → 1 → 3 → 0

next permutation: → 2 → 0 → 1 → 3 · · ·
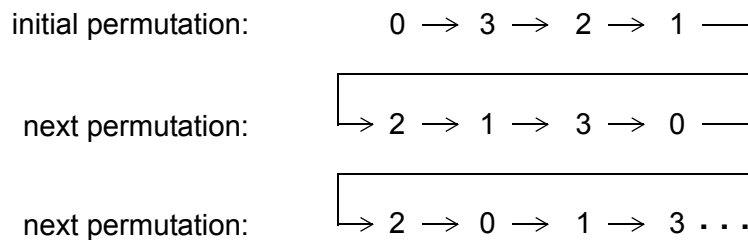
Figure 13-1—Example of randc

The permutation sequence for any given **randc** variable is recomputed whenever the constraints change on that variable or when none of the remaining values in the permutation can satisfy the constraints.

To reduce memory requirements, implementations can impose a limit on the maximum size of a **randc** variable, but it should be no less than 8 bits.

The semantics of random-cyclical variables requires that they be solved before other random variables. A set of constraints that includes both **rand** and **randc** variables shall be solved so that the **randc** variables are solved first, and this can sometimes cause randomize() to fail.

13.4 Constraint blocks

The values of random variables are determined using constraint expressions that are declared using constraint blocks. Constraint blocks are class members, like tasks, functions, and variables. Constraint block names must be unique within a class.

The syntax to declare a constraint block is as follows:

---

constraint_declaration ::=   *// from A.1.9*
     [**static**] **constraint** constraint_identifier constraint_block

constraint_block ::= **{** { constraint_block_item } **}**

constraint_block_item ::=
     solve identifier_list **before** identifier_list **;**
   | constraint_expression

constraint_expression ::=
     expression_or_dist **;**
   | expression **->** constraint_set
   | **if (** expression **)** constraint_set [**else** constraint_set]
   | **foreach (** array_identifier **[** loop_variables **])** constraint_set

constraint_set ::=
     constraint_expression
   | **{** { constraint_expression } **}**

dist_list ::= dist_item { **,** dist_item }

dist_item ::= value_range [dist_weight]

dist_weight ::=
     := expression
   | **:/** expression

constraint_prototype ::= [**static**] **constraint** constraint_identifier **;**

extern_constraint_declaration ::=
     [**static**] **constraint** class_scope constraint_identifier constraint_block

identifier_list ::= identifier { **,** identifier }

expression_or_dist ::= expression [**dist {** dist_list **}**]   *// from A.2.10*

loop_variables ::= [index_variable_identifier] { **,** [index_variable_identifier] }   *// from A.6.8*

Syntax 13-2—Constraint syntax (excerpt from Annex A)

The *constraint_identifier* is the name of the constraint block. This name can be used to enable or disable a constraint using the constraint_mode() method (see 13.8).

The *constraint_block* is a list of expression statements that restrict the range of a variable or define relations between variables. A *constraint_expression* is any SystemVerilog expression or one of the constraint-specific operators, -> and **dist** (see 13.4.4 and 13.4.5).

The declarative nature of constraints imposes the following restrictions on constraint expressions:
   —   Functions are allowed with certain limitations (see 13.4.11).

— Operators with side effects, such as **++** and **--**, are not allowed.

— **randc** variables cannot be specified in ordering constraints (see **solve**…**before** in 13.4.9).

— **dist** expressions cannot appear in other expressions.

13.4.1 External constraint blocks

Constraint block bodies can be declared outside a class declaration, just like external task and function bodies:

```
// class declaration
class XYPair;
   rand integer x, y;
   constraint c;
endclass

// external constraint body declaration
constraint XYPair::c { x < y; }
```

13.4.2 Inheritance

Constraints follow the same general rules for inheritance as class variables, tasks, and functions:

— A constraint in a derived class that uses the same name as a constraint in its parent classes overrides the base class constraints. For example:

```
class A;
   rand integer x;
   constraint c { x < 0; }
endclass

class B extends A;
   constraint c { x > 0; }
endclass
```

An instance of class A constrains x to be less than zero whereas an instance of class B constrains x to be greater than zero. The extended class B overrides the definition of constraint c. In this sense, constraints are treated the same as virtual functions; therefore, casting an instance of B to an A does not change the constraint set.

— The randomize() task is virtual. Accordingly, it treats the class constraints in a virtual manner. When a named constraint is redefined in an extended class, the previous definition is overridden.

13.4.3 Set membership

Constraints support integer value sets and the set membership operator (as defined in 8.19).

Absent any other constraints, all values (either single values or value ranges) have an equal probability of being chosen by the **inside** operator.

The negated form of the **inside** operator denotes that expression lies outside the set: !(expression **inside** { set }).

For example:

```
rand integer x, y, z;
constraint c1 {x inside {3, 5, [9:15], [24:32], [y:2*y], z};}
```

```
    rand integer a, b, c;
    constraint c2 {a inside {b, c};}

    integer fives[4] = '{ 5, 10, 15, 20 };
    rand integer v;
    constraint c3 { v inside fives; }
```

In SystemVerilog, the **inside** operator is bidirectional; thus, the second example above is equivalent to `a == b || a == c`.

13.4.4 Distribution

In addition to set membership, constraints support sets of weighted values called *distributions*. Distributions have two properties: they are a relational test for set membership, and they specify a statistical distribution function for the results.

The syntax to define a distribution expression is as follows:

---

constraint_block ::=   *// from A.1.9*

     ...
    | expression **dist { **dist_list** } ;**

dist_list ::= dist_item { **,** dist_item }

dist_item ::= value_range [dist_weight]

dist_weight ::=
     := expression
    | **:/** expression

dist_item ::=
     value_range **:=** expression
    | value_range **:/** expression

expression_or_dist ::= expression [**dist { **dist_list** } **]   *// from A.2.10*

Syntax 13-3—Constraint distribution syntax (excerpt from Annex A)

The *expression* can be any integral SystemVerilog expression.

The distribution operator **dist** evaluates to true if the value of the expression is contained in the set; otherwise, it evaluates to false.

Absent any other constraints, the probability that the expression matches any value in the list is proportional to its specified weight. If there are constraints on some expressions that cause the distribution weights on these expressions to be not satisfiable, implementations are only required to satisfy the constraints. An exception to this rule is a weight of zero, which is treated as a constraint.

The distribution set is a comma-separated list of integral expressions and ranges. Optionally, each term in the list can have a weight, which is specified using the `:=` or `:/` operators. If no weight is specified for an item, the default weight is `:=` 1. The weight can be any integral SystemVerilog expression.

The `:=` operator assigns the specified weight to the item or, if the item is a range, to every value in the range.

The `:/` operator assigns the specified weight to the item or, if the item is a range, to the range as a whole. If there are n values in the range, the weight of each value is `range_weight / n`.

For example:

```
x dist {100 := 1, 200 := 2, 300 := 5}
```

means x is equal to 100, 200, or 300 with weighted ratio of 1-2-5. If an additional constraint is added that specifies that x cannot be 200,

```
x != 200;
x dist {100 := 1, 200 := 2, 300 := 5}
```

then x is equal to 100 or 300 with weighted ratio of 1-5.

It is easier to think about mixing ratios, such as 1-2-5, than the actual probabilities because mixing ratios do not have to be normalized to 100%. Converting probabilities to mixing ratios is straightforward.

When weights are applied to ranges, they can be applied to each value in the range, or they can be applied to the range as a whole. For example:

```
x dist { [100:102] := 1, 200 := 2, 300 := 5}
```

means x is equal to 100, 101, 102, 200, or 300 with a weighted ratio of 1-1-1-2-5, and

```
x dist { [100:102] :/ 1, 200 := 2, 300 := 5}
```

means x is equal to one of 100, 101, 102, 200, or 300 with a weighted ratio of 1/3-1/3-1/3-2-5.

In general, distributions guarantee two properties: set membership and monotonic weighting. In other words, increasing a weight increases the likelihood of choosing those values.

Limitations:

— A **dist** operation shall not be applied to **randc** variables.
— A **dist** expression requires that expression contain at least one **rand** variable.

13.4.5 Implication

Constraints provide two constructs for declaring conditional (predicated) relations: implication and **if**...**else**.

The implication operator (->) can be used to declare an expression that implies a constraint.

The syntax to define an implication constraint is as follows:

```
constraint_expression ::=                                          // from A.1.9
       ...
     | expression –> constraint_set
```

Syntax 13-4—Constraint implication syntax (excerpt from Annex A)

The *expression* can be any integral SystemVerilog expression.

The boolean equivalent of the implication operator a -> b is (!a || b). This states that if the expression is true, then random numbers generated are constrained by the constraint (or constraint set). Otherwise, the random numbers generated are unconstrained.

The *constraint_set* represents any valid constraint or an unnamed constraint set. If the expression is true, all of the constraints in the constraint set must also be satisfied.

For example:

```
mode == little -> len < 10;
mode == big -> len > 100;
```

In this example, the value of `mode` implies that the value of `len` shall be constrained to less than 10 (`mode == little`), greater than 100 (`mode == big`), or unconstrained (`mode != little` and `mode != big`).

In the example

```
bit [3:0] a, b;
constraint c { (a == 0) -> (b == 1); }
```

both `a` and `b` are 4 bits; therefore, there are 256 combinations of `a` and `b`. Constraint `c` says that `a == 0` implies that `b == 1`, thereby eliminating 15 combinations: {0,0}, {0,2}, … {0,15}. Therefore, the probability that `a == 0` is thus 1/(256-15) or 1/241.

13.4.6 If...else constraints

The **if**...**else** style constraints are also supported.

The syntax to define an **if**...**else** constraint is as follows:

---

constraint_expression ::=   *// from A.1.9*

    ...
    | **if (** expression **)** constraint_set [**else** constraint_set]

Syntax 13-5—If...else constraint syntax (excerpt from Annex A)

The *expression* can be any integral SystemVerilog expression.

The *constraint_set* represents any valid constraint or an unnamed constraint block. If the expression is true, all of the constraints in the first constraint or constraint set must be satisfied; otherwise, all of the constraints in the optional **else** constraint or constraint-block must be satisfied. Constraint sets can be used to group multiple constraints.

The **if...else** style constraint declarations are equivalent to implications

```
if (mode == little)
    len < 10;
else if (mode == big)
    len > 100;
```

which is equivalent to

```
mode == little -> len < 10 ;
mode == big -> len > 100 ;
```

In this example, the value of `mode` implies that the value of `len` is less than 10, greater than 100, or unconstrained.

Just like implication, **if**...**else** style constraints are bidirectional. In the declaration above, the value of mode constrains the value of len, and the value of len constrains the value of mode.

Because the **else** part of an **if**...**else** style constraint declaration is optional, there can be confusion when an **else** is omitted from a nested **if** sequence. This is resolved by always associating the **else** with the closest previous **if** that lacks an **else**. In the example below, the **else** goes with the inner **if**, as shown by indentation:

```
if (mode != big)
    if (mode == little)
        len < 10;
    else // the else applies to preceding if
        len > 100;
```

13.4.7 Iterative constraints

Iterative constraints allow arrayed variables to be constrained in a parameterized manner using loop variables and indexing expressions.

The syntax to define an iterative constraint is as follows:

constraint_expression ::=	*// from A.1.9*
...	
| **foreach (** array_identifier **[** loop_variables **])** constraint_set	
loop_variables ::= [index_variable_identifier] { **,** [index_variable_identifier] }	*// from A.6.8*

Syntax 13-6—Foreach iterative constraint syntax (excerpt from Annex A)

The **foreach** construct specifies iteration over the elements of an array. Its argument is an identifier that designates any type of array (fixed-size, dynamic, associative, or queue) followed by a list of loop variables enclosed in square brackets. Each loop variable corresponds to one of the dimensions of the array.

For example:

```
class C;
    rand byte A[] ;

    constraint C1 { foreach ( A [ i ] ) A[i] inside {2,4,8,16}; }
    constraint C2 { foreach ( A [ j ] ) A[j] > 2 * j; }
endclass
```

C1 constrains each element of the array A to be in the set [2,4,8,16]. C2 constrains each element of the array A to be greater than twice its index.

The number of loop variables must not exceed the number of dimensions of the array variable. The scope of each loop variable is the **foreach** constraint construct, including its *constraint_set*. The type of each loop variable is implicitly declared to be consistent with the type of array index. An empty loop variable indicates no iteration over that dimension of the array. As with default arguments, a list of commas at the end can be omitted; thus, **foreach**(arr [j]) is a shorthand for **foreach**(arr [j, , , ,]). It shall be an error for any loop variable to have the same identifier as the array.

The mapping of loop variables to array indexes is determined by the dimension cardinality, as described in 22.6.

```
//     1  2  3          3   4     1   2       -> Dimension numbers
int A [2][3][4];    bit [3:0][2:1] B [5:1][4];

foreach( A [ i, j, k ] ) ...
foreach( B [ q, r, , s ] ) ...
```

The first **foreach** causes i to iterate from 0 to 1, j from 0 to 2, and k from 0 to 3. The second **foreach** causes q to iterate from 5 to 1, r from 0 to 3, and s from 2 to 1.

Iterative constraints can include predicates. For example:

```
class C;
   rand int A[] ;

   constraint c1 { A.size inside {[1:10]}; }
   constraint c2 { foreach ( A[ k ] ) (k < A.size - 1) -> A[k + 1] > A[k]; }
endclass
```

The first constraint, c1, constrains the size of the array A to be between 1 and 10. The second constraint, c2, constrains each array value to be greater than the preceding one, i.e., an array sorted in ascending order.

Within a **foreach**, predicate expressions involving only constants, state variables, object handle comparisons, loop variables, or the size of the array being iterated behave as guards against the creation of constraints, and not as logical relations. For example, the implication in constraint c2 above involves only a loop variable and the size of the array being iterated; thus, it allows the creation of a constraint only when k < A.size() - 1, which in this case prevents an out-of-bounds access in the constraint. Guards are described in more detail in 13.4.12.

Index expressions can include loop variables, constants, and state variables. Invalid or out or bound array indexes are not automatically eliminated; users must explicitly exclude these indexes using predicates.

The size method of a dynamic or associative array can be used to constrain the size of the array (see constraint c1 above). If an array is constrained by both size constraints and iterative constraints, the size constraints are solved first, and the iterative constraints next. As a result of this implicit ordering between size constraints and iterative constraints, the size method shall be treated as a state variable within the **foreach** block of the corresponding array. For example, the expression A.size is treated as a random variable in constraint c1 and as a state variable in constraint c2. This implicit ordering can cause the solver to fail in some situations.

13.4.8 Global constraints

When an object member of a class is declared **rand**, all of its constraints and random variables are randomized simultaneously along with the other class variables and constraints. Constraint expressions involving random variables from other objects are called *global constraints* (see Figure 13-2).

```
class A;              // leaf node
   rand bit [7:0] v;
endclass

class B extends A; // heap node
   rand A left;
   rand A right;

   constraint heapcond {left.v <= v; right.v <= v;}
endclass
```
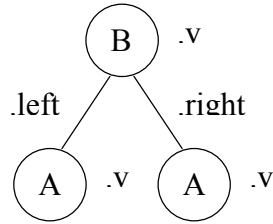
Figure 13-2—Global constraints

This example uses global constraints to define the legal values of an ordered binary tree. Class A represents a leaf node with an 8-bit value v. Class B extends class A and represents a heap node with value v, a left sub-tree, and a right subtree. Both subtrees are declared as **rand** in order to randomize them at the same time as other class variables. The constraint block named heapcond has two global constraints relating the left and right subtree values to the heap node value. When an instance of class B is randomized, the solver simultaneously solves for B and its left and right children, which in turn can be leaf nodes or more heap nodes.

The following rules determine which objects, variables, and constraints are to be randomized:

a)   First, determine the set of objects that are to be randomized as a whole. Starting with the object that invoked the randomize() method, add all objects that are contained within it, are declared **rand**, and are active (see rand_mode in 13.7). The definition is recursive and includes all of the active random objects that can be reached from the starting object. The objects selected in this step are referred to as the *active random objects*.

b)   Second, select all of the active constraints from the set of active random objects. These are the constraints that are applied to the problem.

c)   Third, select all of the active random variables from the set of active random objects. These are the variables that are to be randomized. All other variable references are treated as state variables, whose current value is used as a constant.

13.4.9 Variable ordering

The solver must assure that the random values are selected to give a uniform value distribution over legal value combinations (that is, all combinations of legal values have the same probability of being the solution). This important property guarantees that all legal value combinations are equally probable, which allows randomization to better explore the whole design space.

Sometimes, however, it is desirable to force certain combinations to occur more frequently. Consider the case where a 1-bit control variable s constrains a 32-bit data value d:

```
class B;
   rand bit s;
   rand bit [31:0] d;

   constraint c { s -> d == 0; }
endclass
```

The constraint c says "s implies d equals zero". Although this reads as if s determines d, in fact s and d are determined together. There are 2^{33} possible combinations of {s,d}, but s is only true for {1,0}. Thus, the probability that s is true is $1/2^{33}$, which is practically zero.

The constraints provide a mechanism for ordering variables so that `s` can be chosen independently of `d`. This mechanism defines a partial ordering on the evaluation of variables and is specified using the `solve` keyword.

```
class B;
    rand bit s;
    rand bit [31:0] d;
    constraint c { s -> d == 0; }
    constraint order { solve s before d; }
endclass
```

In this case, the order constraint instructs the solver to solve for `s` before solving for `d`. The effect is that `s` is now chosen true with 50% probability, and then `d` is chosen subject to the value of `s`. Accordingly, `d == 0` shall occur 50% of the time, and `d != 0` shall occur for the other 50%.

Variable ordering can be used to force selected corner cases to occur more frequently than they would otherwise. However, a "`solve`...`before`..." constraint does not change the solution space and, therefore, cannot cause the solver to fail.

The syntax to define variable order in a constraint block is as follows:

constraint_block_item ::= *// from A.1.9* **solve** identifier_list **before** identifier_list **;** 	constraint_expression

Syntax 13-7—Solve...before constraint ordering syntax (excerpt from Annex A)

The `solve` and `before` each take a comma-separated list of integral variables or array elements.

The following restrictions apply to variable ordering:
— Only random variables are allowed, that is, they must be `rand`.
— `randc` variables are not allowed. `randc` variables are always solved before any other.
— The variables must be integral values.
— A constraint block can contain both regular value constraints and ordering constraints.
— There must be no circular dependencies in the ordering, such as "solve a before b" combined with "solve b before a".
— Variables that are not explicitly ordered shall be solved with the last set of ordered variables. These values are deferred until as late as possible to assure a good distribution of values.
— Variables that are partially ordered shall be solved with the latest set of ordered variables so that all ordering constraints are met. These values are deferred until as late as possible to assure a good distribution of values.
— Variables can be solved in an order that is not consistent with the ordering constraints, provided that the outcome is the same. An example situation where this might occur is as follows:

```
        x == 0;
        x < y;
        solve y before x;
```

In this case, because `x` has only one possible assignment (0), `x` can be solved for before `y`. The constraint solver can use this flexibility to speed up the solving process.

13.4.10 Static constraint blocks

A constraint block can be defined as static by including the **static** keyword in its definition.

The syntax to declare a static constraint block is as follows:

---

constraint_declaration ::=   *// from A.1.9*
     [**static**] **constraint** constraint_identifier constraint_block

---

Syntax 13-8—Static constraint syntax (excerpt from Annex A)

If a constraint block is declared as **static**, then calls to constraint_mode() shall affect all instances of the specified constraint in all objects. Thus, if a static constraint is set to OFF, it is off for all instances of that particular class.

13.4.11 Functions in constraints

Some properties are unwieldy or impossible to express in a single expression. For example, the natural way to compute the number of ones in a packed array uses a loop:

```
function int count_ones ( bit [9:0] w );
   for( count_ones = 0; w != 0; w = w >> 1 )
      count_ones += w & 1'b1;
endfunction
```

Such a function could be used to constrain other random variables to the number of 1 bits:

```
constraint C1  { length == count_ones( v ) ; }
```

Without the ability to call a function, this constraint requires the loop to be unrolled and expressed as a sum of the individual bits:

```
constraint C2
{
    length == ((v>>9)&1) + ((v>>8)&1) + ((v>>7)&1) + ((v>>6)&1) + ((v>>5)&1) +
              ((v>>4)&1) + ((v>>3)&1) + ((v>>2)&1) + ((v>>1)&1) + ((v>>0)&1);
}
```

Unlike the count_ones function, more complex properties, which require temporary state or unbounded loops, may be impossible to convert into a single expression. The ability to call functions, thus, enhances the expressive power of the constraint language and reduces the likelihood of errors. The two constraints, C1 and C2, from above are not completely equivalent; C2 is bidirectional (length can constrain v and vice versa), whereas C1 is not.

To handle these common cases, SystemVerilog allows constraint expressions to include function calls, but it imposes certain semantic restrictions:

— Functions that appear in constraint expressions cannot contain **output** or **ref** arguments (**const ref** are allowed).

— Functions that appear in constraint expressions should be automatic (or preserve no state information) and have no side effects.

— Functions that appear in constraints cannot modify the constraints, for example, calling rand_mode or constraint_mode methods.

— Functions shall be called before constraints are solved, and their return values shall be treated as state variables.

— Random variables used as function arguments shall establish an implicit variable ordering or priority. Constraints that include only variables with higher priority are solved before other, lower priority constraints. Random variables solved as part of a higher priority set of constraints become state variables to the remaining set of constraints. For example:

```
class B;
    rand int x, y;
    constraint C { x <= F(y); }
    constraint D { y inside { 2, 4, 8 } ; }
endclass
```

forces `y` to be solved before `x`. Thus, constraint `D` is solved separately before constraint `C`, which uses the values of `y` and `F(y)` as state variables. In SystemVerilog, the behavior for variable ordering implied by function arguments differs from the behavior for ordering specified using the "**solve**...**before**..." constraint; function argument variable ordering subdivides the solution space thereby changing it. Because constraints on higher priority variables are solved without considering lower priority constraints at all, this subdivision can cause the overall constraints to fail. Within each prioritized set of constraints, cyclical (**randc**) variables are solved first.

— Circular dependencies created by the implicit variable ordering shall result in an error.

— Function calls in active constraints are executed an unspecified number of times (at least once) in an unspecified order.

13.4.12 Constraint guards

Constraint guards are predicate expressions that function as guards against the creation of constraints, and not as logical relations to be satisfied by the solver. These predicate expressions are evaluated before the constraints are solved and are characterized by involving only the following items:

— Constants

— State variables

— Object handle comparisons (comparisons between two handles or a handle and the constant **null**)

In addition to the above, iterative constraints (see 13.4.7) also consider loop variables and the size of the array being iterated as state variables.

Treating these predicate expressions as constraint guards prevents the solver from generating evaluation errors, thereby failing on some seemingly correct constraints. This enables users to write constraints that avoid errors due to nonexistent object handles or array indices out of bounds. For example, the sort constraint of the singly linked list, `SList`, shown below is intended to assign a random sequence of numbers that is sorted in ascending order. However, the constraint expression will fail on the last element when `next.n` results in an evaluation error due to a nonexistent handle.

```
class SList;
    rand int n;
    rand Slist next;

    constraint sort { n < next.n; }
endclass
```

The error condition above can be avoided by writing a predicate expression to guard against that condition:

```
constraint sort { if( next != null ) n < next.n; }
```

In the sort constraint above, the `if` prevents the creation of a constraint when next == **null**, which in this case avoids accessing a nonexistent object. Both implication (`->`) and `if`...`else` can be used as guards.

Guard expressions can themselves include subexpressions that result in evaluation errors (e.g., null references), and they are also guarded from generating errors. This logical sifting is accomplished by evaluating predicate subexpressions using the following 4-state representation:

— 0 FALSE   Subexpression evaluates to FALSE.

— 1 TRUE   Subexpression evaluates to TRUE.

— E ERROR   Subexpression causes an evaluation error.

— R RANDOM   Expression includes random variables and cannot be evaluated.

Every subexpression within a predicate expression is evaluated to yield one of the above four values. The subexpressions are evaluated in an arbitrary order, and the result of that evaluation plus the logical operation define the outcome in the alternate 4-state representation. A conjunction (`&&`), disjunction (`||`), or negation (`!`) of subexpressions can include some (perhaps all) guard subexpressions. The following rules specify the resulting value for the guard:

— Conjunction (`&&`): If any one of the subexpressions evaluates to FALSE, then the guard evaluates to FALSE. If any one subexpression evaluates to ERROR, then the guard evaluates to ERROR. Otherwise, the guard evaluates to TRUE.

  — If the guard evaluates to FALSE, then the constraint is eliminated.

  — If the guard evaluates to TRUE, then a (possibly conditional) constraint is generated.

  — If the guard evaluates to ERROR, then an error is generated and randomize fails.

— Disjunction (`||`): If any one of the subexpressions evaluates to TRUE, then the guard evaluates to TRUE. If any one subexpression evaluates to ERROR, then the guard evaluates to ERROR. Otherwise, the guard evaluates to FALSE.

  — If the guard evaluates to FALSE, then a (possibly conditional) constraint is generated.

  — If the guard evaluates to TRUE, then an unconditional constraint is generated.

  — If the guard evaluates to ERROR, then an error is generated and randomize fails.

— Negation (`!`): If the subexpression evaluates to ERROR, then the guard evaluates to ERROR. Otherwise, if the subexpression evaluates to TRUE or FALSE, then the guard evaluates to FALSE or TRUE, respectively.

These rules are codified by the truth tables shown in Figure 13-3.

&&	0	1	E	R
0	0	0	0	0
1	0	1	E	R
E	0	E	E	E
R	0	R	E	R

Conjunction

||	0	1	E	R
0	0	1	E	R
1	1	1	1	1
E	E	1	E	E
R	R	1	E	R

Disjunction

!	
0	1
1	0
E	E
R	R

Negation

Figure 13-3—Trust tables for conjunction, disjunction, and negation rules

These rules are applied recursively until all subexpressions are evaluated. The final value of the evaluated predicate expression determines the outcome as follows:

— If the result is TRUE, then an unconditional constraint is generated.
— If the result is FALSE, then the constraint is eliminated and can generate no error.
— If the result is ERROR, then an unconditional error is generated and the constraint fails.
— If the final result of the evaluation is RANDOM, then a conditional constraint is generated.

When the final value is RANDOM, a traversal of the predicate expression tree is needed to collect all conditional guards that evaluate to RANDOM. When the final value is ERROR, a subsequent traversal of the expression tree is not required, allowing implementations to issue only one error.

Example 1:

```
class D;
    int x;
endclass

class C;
    rand int x, y;
    D a, b;
    constraint c1 { (x < y || a.x > b.x || a.x == 5 ) -> x+y == 10; }
endclass
```

In Example 1, the predicate subexpressions are (x < y), (a.x > b.x), and (a.x == 5), which are all connected by disjunction. Some possible cases are as follows:

— Case 1: a is non-**null**, b is **null**, a.x is 5.

  Because (a.x==5) is true, the fact that b.x generates an error does not result in an error.
  The unconditional constraint (x+y == 10) is generated.
— Case 2: a is null.

  This always results in error, irrespective of the other conditions.
— Case 3: a is non-**null**, b is non-**null**, a.x is 10, b.x is 20.

  All the guard subexpressions evaluate to FALSE.
  The conditional constraint (x<y) -> (x+y == 10) is generated.

Example 2:

```
class D;
    int x;
endclass

class C;
    rand int x, y;
    D a, b;
    constraint c1 { (x < y && a.x > b.x && a.x == 5 ) -> x+y == 10; }
endclass
```

In Example 2, the predicate subexpressions are (x < y), (a.x > b.x), and (a.x == 5), which are all connected by conjunction. Some possible cases are as follows:

— Case 1: a is non-**null**, b is **null**, a.x is 6.

  Because (a.x==5) is false, the fact that b.x generates an error does not result in an error.
  The constraint is eliminated.

— Case 2: `a` is **null**

This always results in error, irrespective of the other conditions.

— Case 3: `a` is non-**null**, `b` is non-**null**, `a.x` is 5, `b.x` is 2.

All the guard subexpressions evaluate to TRUE, producing constraint `(x<y) -> (x+y == 10)`.

Example 3:

```
class D;
   int x;
endclass

class C;
   rand int x, y;
   D a, b;
   constraint c1 { (x < y && (a.x > b.x || a.x ==5)) -> x+y == 10; }
endclass
```

In Example 3, the predicate subexpressions are `(x < y)` and `(a.x > b.x || a.x == 5)`, which are connected by disjunction. Some possible cases are as follows:

— Case 1: `a` is non-**null**, `b` is **null**, `a.x` is 5.

The guard expression evaluates to `(ERROR || a.x==5)`, which evaluates to `(ERROR || TRUE)`
The guard subexpression evaluates to TRUE.
The conditional constraint `(x<y) -> (x+y == 10)` is generated.

— Case 2: `a` is non-**null**, `b` is **null**, `a.x` is 8.

The guard expression evaluates to `(ERROR || FALSE)` and generates an error.

— Case 3: `a` is **null**

This always results in error, irrespective of the other conditions.

— Case 4: `a` is non-**null**, `b` is non-**null**, `a.x` is 5, `b.x` is 2.

All the guard subexpressions evaluate to TRUE.
The conditional constraint `(x<y) -> (x+y == 10)` is generated.

13.5 Randomization methods

13.5.1 Randomize()

Variables in an object are randomized using the `randomize()` class method. Every class has a built-in `randomize()` virtual method, declared as follows:

```
virtual function int randomize();
```

The `randomize()` method is a virtual function that generates random values for all the active random variables in the object, subject to the active constraints.

The `randomize()` method returns 1 if it successfully sets all the random variables and objects to valid values; otherwise, it returns 0.

Example:

```
class SimpleSum;
   rand bit [7:0] x, y, z;
   constraint c {z == x + y;}
endclass
```

This class definition declares three random variables, `x`, `y`, and `z`. Calling the `randomize()` method shall randomize an instance of class `SimpleSum`:

```
SimpleSum p = new;
int success = p.randomize();
if (success == 1 ) ...
```

Checking the return status can be necessary because the actual value of state variables or addition of constraints in derived classes can render seemingly simple constraints unsatisfiable.

13.5.2 Pre_randomize() and post_randomize()

Every class contains `pre_randomize()` and `post_randomize()` methods, which are automatically called by `randomize()` before and after computing new random values.

The prototype for the `pre_randomize()` method is as follows:

```
function void pre_randomize();
```

The prototype for the `post_randomize()` method is as follows:

```
function void post_randomize();
```

When `obj.randomize()` is invoked, it first invokes `pre_randomize()` on `obj` and also all of its random object members that are enabled. After the new random values are computed and assigned, `randomize()` invokes `post_randomize()` on `obj` and also all of its random object members that are enabled.

Users can override the `pre_randomize()` in any class to perform initialization and set preconditions before the object is randomized. If the class is a derived class and no user-defined implementation of `pre_randomize()` exists, then `pre_randomize()` will automatically invoke `super.pre_randomize()`.

Users can override the `post_randomize()` in any class to perform cleanup, print diagnostics, and check post-conditions after the object is randomized. If the class is a derived class and no user-defined implementation of `post_randomize()` exists, then `post_randomize()` will automatically invoke `super.post_randomize()`.

If these methods are overridden, they must call their associated parent class methods; otherwise, their pre- and post-randomization processing steps shall be skipped.

The `pre_randomize()` and `post_randomize()` methods are not virtual. However, because they are automatically called by the `randomize()` method, which is virtual, they appear to behave as virtual methods.

13.5.3 Behavior of randomization methods

— Random variables declared as static are shared by all instances of the class in which they are declared. Each time the `randomize()` method is called, the variable is changed in every class instance.

— If `randomize()` fails, the constraints are infeasible, and the random variables retain their previous values.

— If `randomize()` fails, `post_randomize()` is not called.

— The `randomize()` method is built-in and cannot be overridden.

— The `randomize()` method implements object random stability. An object can be seeded by calling its `srandom()` method (see 13.12.3).

— The built-in methods `pre_randomize()` and `post_randomize()` are functions and cannot block.

13.6 In-line constraints—randomize() with

By using the `randomize()`...**with** construct, users can declare in-line constraints at the point where the `randomize()` method is called. These additional constraints are applied along with the object constraints.

The syntax for `randomize()`...**with** is as follows:

inline_constraint _declaration ::=   *// not in Annex A*
    class_variable_identifier **. randomize** [**(** [variable_identifier_list | **null**] **)**]
      with constraint_block

Syntax 13-9—In-line constraint syntax (not in Annex A)

The *class_variable_identifier* is the name of an instantiated object.

The unnamed *constraint_block* contains additional in-line constraints to be applied along with the object constraints declared in the class.

For example:

```
class SimpleSum
   rand bit [7:0] x, y, z;
   constraint c {z == x + y;}
endclass

task InlineConstraintDemo(SimpleSum p);
   int success;
   success = p.randomize() with {x < y;};
endtask
```

This is the same example used before; however, `randomize()`...**with** is used to introduce an additional constraint that `x < y`.

The `randomize()`...**with** construct can be used anywhere an expression can appear. The constraint block following **with** can define all of the same constraint types and forms as would otherwise be declared in a class.

The `randomize()`...**with** constraint block can also reference local variables and task and function arguments, eliminating the need for mirroring a local state as member variables in the object class. The scope for variable names in a constraint block, from inner to outer, is `randomize()`...**with** object class, automatic and local variables, task and function arguments, class variables, and variables in the enclosing scope. The `randomize()`...**with** class is brought into scope at the innermost nesting level.

In the example below, the `randomize()`...**with** class is Foo.

```
class Foo;
   rand integer x;
endclass

class Bar;
```

```
        integer x;
        integer y;

        task doit(Foo f, integer x, integer z);
            int result;
            result = f.randomize() with {x < y + z;};
        endtask
    endclass
```

In the `f.randomize()` **with** constraint block, x is a member of class `Foo` and hides the x in class `Bar`. It also hides the x argument in the `doit()` task. y is a member of `Bar`. z is a local argument.

13.7 Disabling random variables with rand_mode()

The `rand_mode()` method can be used to control whether a random variable is active or inactive. When a random variable is inactive, it is treated the same as if it had not been declared **rand** or **randc**. Inactive variables are not randomized by the `randomize()` method, and their values are treated as state variables by the solver. All random variables are initially active.

The syntax for the `rand_mode()` method is as follows:

```
    task object[.random_variable]::rand_mode( bit on_off );
```

or

```
    function int object.random_variable::rand_mode();
```

The *object* is any expression that yields the object handle in which the random variable is defined.

The *random_variable* is the name of the random variable to which the operation is applied. If it is not specified (only allowed when called as a task), the action is applied to all random variables within the specified object.

When called as a task, the argument to the `rand_mode` method determines the operation to be performed as shown in Table 13-1:

Table 13-1—rand_mode argument

Value	Meaning	Description
0	OFF	Sets the specified variables to inactive so that they are not randomized on subsequent calls to the `randomize()` method.
1	ON	Sets the specified variables to active so that they are randomized on subsequent calls to the `randomize()` method.

For unpacked array variables, `random_variable` can specify individual elements using the corresponding index. Omitting the index results in all the elements of the array being affected by the call.

For unpacked structure variables, `random_variable` can specify individual members using the corresponding member. Omitting the member results in all the members of the structure being affected by the call.

If the random variable is an object handle, only the mode of the variable is changed, not the mode of random variables within that object (see global constraints in 13.4.8).

A compiler error shall be issued if the specified variable does not exist within the class hierarchy or it exists but is not declared as **rand** or **randc**.

When called as a function, `rand_mode()` returns the current active state of the specified random variable. It returns 1 if the variable is active (ON) and 0 if the variable is inactive (OFF).

The function form of `rand_mode()` only accepts singular variables; thus, if the specified variable is an unpacked array, a single element must be selected via its index.

Example:

```
class Packet;
    rand integer source_value, dest_value;
    ... other declarations
endclass

int ret;
Packet packet_a = new;
// Turn off all variables in object
packet_a.rand_mode(0);

// ... other code
// Enable source_value
packet_a.source_value.rand_mode(1);

ret = packet_a.dest_value.rand_mode();
```

This example first disables all random variables in the object `packet_a` and then enables only the `source_value` variable. Finally, it sets the `ret` variable to the active status of variable `dest_value`.

The `rand_mode()` method is built-in and cannot be overridden.

13.8 Controlling constraints with constraint_mode()

The `constraint_mode()` method can be used to control whether a constraint is active or inactive. When a constraint is inactive, it is not considered by the `randomize()` method. All constraints are initially active.

The syntax for the `constraint_mode()` method is as follows:

```
task object[.constraint_identifier]::constraint_mode( bit on_off );
```

or

```
function int object.constraint_identifier::constraint_mode();
```

The *object* is any expression that yields the object handle in which the constraint is defined.

The *constraint_identifier* is the name of the constraint block to which the operation is applied. The constraint name can be the name of any constraint block in the class hierarchy. If no constraint name is specified (only allowed when called as a task), the operation is applied to all constraints within the specified object.

When called as a task, the argument to the `constraint_mode` task method determines the operation to be performed as shown in Table 13-2:

Table 13-2—constraint_mode argument

Value	Meaning	Description
0	OFF	Sets the specified constraint block to inactive so that it is not enforced by subsequent calls to the `randomize()` method.
1	ON	Sets the specified constraint block to active so that it is considered on subsequent calls to the `randomize()` method.

A compiler error shall be issued if the specified constraint block does not exist within the class hierarchy.

When called as a function, `constraint_mode()` returns the current active state of the specified constraint block. It returns 1 if the constraint is active (ON) and 0 if the constraint is inactive (OFF).

Example:

```
class Packet;
   rand integer source_value;
   constraint filter1 { source_value > 2 * m; }
endclass

function integer toggle_rand( Packet p );
   if ( p.filter1.constraint_mode() )
      p.filter1.constraint_mode(0);
   else
      p.filter1.constraint_mode(1);

   toggle_rand = p.randomize();
endfunction
```

In this example, the `toggle_rand` function first checks the current active state of the constraint `filter1` in the specified `Packet` object `p`. If the constraint is active, the function deactivates it; if it is inactive, the function activates it. Finally, the function calls the randomize method to generate a new random value for variable `source_value`.

The `constraint_mode()` method is built-in and cannot be overridden.

13.9 Dynamic constraint modification

There are several ways to dynamically modify randomization constraints:

— Implication and **if**...**else** style constraints allow declaration of predicated constraints.

— Constraint blocks can be made active or inactive using the `constraint_mode()` built-in method. Initially, all constraint blocks are active. Inactive constraints are ignored by the `randomize()` function.

— Random variables can be made active or inactive using the `rand_mode()` built-in method. Initially, all **rand** and **randc** variables are active. Inactive variables are ignored by the `randomize()` function.

—   The weights in a **dist** constraint can be changed, affecting the probability that particular values in the set are chosen.

13.10 In-line random variable control

The `randomize()` method can be used to temporarily control the set of random and state variables within a class instance or object. When the randomize method is called with no arguments, it behaves as described in the previous subclauses, that is, it assigns new values to all random variables in an object—those declared as **rand** or **randc**—so that all of the constraints are satisfied. When randomize is called with arguments, those arguments designate the complete set of random variables within that object; all other variables in the object are considered state variables. For example, consider the following class and calls to randomize:

```
class CA;
   rand byte x, y;
   byte v, w;

   constraint c1 { x < v && y > w );
endclass


CA a = new;

a.randomize();          // random variables: x, y  state variables: v, w
a.randomize( x );       // random variables: x     state variables: y, v, w
a.randomize( v, w );    // random variables: v, w  state variables: x, y
a.randomize( w, x );    // random variables: w, x  state variables: y, v
```

This mechanism controls the set of active random variables for the duration of the call to randomize, which is conceptually equivalent to making a set of calls to the `rand_mode()` method to disable or enable the corresponding random variables. Calling `randomize()` with arguments allows changing the random mode of any class property, even those not declared as **rand** or **randc**. This mechanism, however, does not affect the cyclical random mode; it cannot change a nonrandom variable into a cyclical random variable (**randc**) and cannot change a cyclical random variable into a noncyclical random variable (change from **randc** to **rand**).

The scope of the arguments to the randomize method is the object class. Arguments are limited to the names of properties of the calling object; expressions are not allowed. The random mode of local class members can only be changed when the call to randomize has access to those properties, that is, within the scope of the class in which the local members are declared.

13.10.1 In-line constraint checker

Normally, calling the randomize method of a class that has no random variables causes the method to behave as a checker. In other words, it assigns no random values and only returns a status: 1 if all constraints are satisfied and 0 otherwise. The in-line random variable control mechanism can also be used to force the `randomize()` method to behave as a checker.

The randomize method accepts the special argument **null** to indicate no random variables for the duration of the call. In other words, all class members behave as state variables. This causes the randomize method to behave as a checker instead of a generator. A checker evaluates all constraints and simply returns 1 if all constraints are satisfied and 0 otherwise. For example, if class CA defined above executes the following call:

```
success = a.randomize( null );  // no random variables
```

then the solver considers all variables as state variables and only checks whether the constraint is satisfied, namely, that the relation (x < v && y > w) is true using the current values of x, y, v, and w.

13.11 Randomization of scope variables—std::randomize()

The built-in class randomize method operates exclusively on class member variables. Using classes to model the data to be randomized is a powerful mechanism that enables the creation of generic, reusable objects containing random variables and constraints that can be later extended, inherited, constrained, over-ridden, enabled, disabled, and merged with or separated from other objects. The ease with which classes and their associated random variables and constraints can be manipulated makes classes an ideal vehicle for describing and manipulating random data and constraints. However, some less-demanding problems that do not require the full flexibility of classes can use a simpler mechanism to randomize data that do not belong to a class. The scope randomize function, `std::randomize()`, enables users to randomize data in the current scope without the need to define a class or instantiate a class object.

The syntax of the scope randomize function is as follows:

scope_randomize ::=   *// not in Annex A*
     [**std::**] **randomize (** [variable_identifier_list] **)** [**with** constraint_block]

Syntax 13-10—Scope randomize function syntax (not in Annex A)

The scope randomize function behaves exactly the same as a class randomize method, except that it operates on the variables of the current scope instead of class member variables. Arguments to this function specify the variables that are to be assigned random values, i.e., the random variables.

For example:

```
module stim;
   bit [15:0] addr;
   bit [31:0] data;

   function bit gen_stim();
      bit success, rd_wr;

      success = randomize( addr, data, rd_wr );  // call std::randomize
      return rd_wr ;
   endfunction

   ...
endmodule
```

The function `gen_stim` calls `std::randomize()` with three variables as arguments: `addr`, `data`, and `rd_wr`. Thus, `std::randomize()` assigns new random variables to the variables that are visible in the scope of the `gen_stim` function. In the preceding example, `addr` and `data` have module scope, whereas `rd_wr` has scope local to the function. The preceding example can also be written using a class:

```
class stimc;
   rand bit [15:0] addr;
   rand bit [31:0] data;
   rand bit rd_wr;
endclass

function bit gen_stim( stimc p );
   bit success;
   success = p.randomize();
   addr = p.addr;
   data = p.data;
   return p.rd_wr;
```

```
    endfunction
```

However, for this simple application, the scope randomize function leads to a straightforward implementation.

The scope randomize function returns 1 if it successfully sets all the random variables to valid values; otherwise, it returns 0. If the scope randomize function is called with no arguments, then it behaves as a checker and simply returns status.

13.11.1 Adding constraints to scope variables—std::randomize() with

The `std::randomize()` **with** form of the scope randomize function allows users to specify random constraints to be applied to the local scope variables. When specifying constraints, the arguments to the scope randomize function become random variables; all other variables are considered state variables.

```
    task stimulus( int length );
        int a, b, c, success;

        success = std::randomize( a, b, c ) with { a < b ; a + b < length ; };
        ...
        success = std::randomize( a, b ) with { b - a > length ; };
        ...
    endtask
```

The task stimulus above calls `std::randomize` twice resulting in two sets of random values for its local variables a, b, and c. In the first call, variables a and b are constrained so that variable a is less than b and their sum is less than the task argument length, which is designated as a state variable. In the second call, variables a and b are constrained so that their difference is greater than the state variable length.

13.12 Random number system functions and methods

13.12.1 $urandom

The system function `$urandom` provides a mechanism for generating pseudo-random numbers. The function returns a new 32-bit random number each time it is called. The number shall be unsigned.

The syntax for `$urandom` is as follows:

```
    function int unsigned $urandom [ (int seed ) ] ;
```

The `seed` is an optional argument that determines the sequence of random numbers generated. The seed can be any integral expression. The random number generator (RNG) shall generate the same sequence of random numbers every time the same seed is used.

The RNG is deterministic. Each time the program executes, it cycles through the same random sequence. This sequence can be made nondeterministic by seeding the `$urandom` function with an extrinsic random variable, such as the time of day.

For example:

```
    bit [64:1] addr;

    $urandom( 254 );                // Initialize the generator
    addr = {$urandom, $urandom }; // 64-bit random number
    number = $urandom & 15;       // 4-bit random number
```

The $urandom function is similar to the $random system function, with two exceptions: $urandom returns unsigned numbers and is automatically thread stable (see 13.13.2).

13.12.2 $urandom_range()

The $urandom_range() function returns an unsigned integer within a specified range.

The syntax for $urandom_range() is as follows:

```
function int unsigned $urandom_range(  int unsigned maxval,
                                       int unsigned minval = 0 );
```

The function shall return an unsigned integer in the range of maxval ... minval.

Example:

```
val = $urandom_range(7,0);
```

If minval is omitted, the function shall return a value in the range of maxval ... 0.

Example:

```
val = $urandom_range(7);
```

If maxval is less than minval, the arguments are automatically reversed so that the first argument is larger than the second argument.

Example:

```
val = $urandom_range(0,7);
```

All of the three previous examples produce a value in the range of 0 to 7, inclusive.

$urandom_range() is automatically thread stable (see 13.13.2).

13.12.3 Srandom()

The srandom() method allows manually seeding the RNG of objects or threads. The RNG of a process can be seeded using the srandom() method of the process (see 11.9).

The prototype of the srandom() method is as follows:

```
function void srandom( int seed );
```

The srandom() method initializes an object's RNG using the value of the given seed.

13.12.4 Get_randstate()

The get_randstate() method retrieves the current state an object's RNG. The state of the RNG associated with a process is retrieved using the get_randstate() method of the process (see 11.9).

The prototype of the get_randstate() method is as follows:

```
function string get_randstate();
```

The `get_randstate()` method returns a copy of the internal state of the RNG associated with the given object.

The RNG state is a string of unspecified length and format. The length and contents of the string are implementation dependent.

13.12.5 Set_randstate()

The `set_randstate()` method sets the state of an object's RNG. The state of the RNG associated with a process is set using the `set_randstate()` method of the process (see 11.9).

The prototype of the `set_randstate()` method is as follows:

```
function void set_randstate( string state );
```

The `set_randstate()` method copies the given state into the internal state of an object's RNG.

The RNG state is a string of unspecified length and format. Calling `set_randstate()` with a string value that was not obtained from `get_randstate()`, or from a different implementation of `get_randstate()`, is undefined.

13.13 Random stability

The RNG is localized to threads and objects. Because the sequence of random values returned by a thread or object is independent of the RNG in other threads or objects, this property is called *random stability*. Random stability applies to the following:

— The system randomization calls, `$urandom()` and `$urandom_range()`
— The object and process random seeding method, `srandom()`
— The object randomization method, `randomize()`

Testbenches with this feature exhibit more stable RNG behavior in the face of small changes to the user code. Additionally, it enables more precise control over the generation of random values by manually seeding threads and objects.

13.13.1 Random stability properties

Random stability encompasses the following properties:

— *Initialization RNG*. Each module instance, interface instance, program instance, and package has an initialization RNG. Each initialization RNG is seeded with the default seed. The default seed is an implementation-dependent value. An initialization RNG shall be used in the creation of static threads and static initializers (see the following bullets).

— *Thread stability*. Each thread has an independent RNG for all randomization system calls invoked from that thread. When a new dynamic thread is created, its RNG is seeded with the next random value from its parent thread. This property is called *hierarchical seeding*. When a static thread is created, its RNG is seeded with the next value from the initialization RNG of the module instance, interface instance, program instance, or package containing the thread declaration.

Program and thread stability is guaranteed as long as thread creation and random number generation are done in the same order as before. When adding new threads to an existing test, they can be added at the end of a code block in order to maintain random number stability of previously created work.

— *Object stability*. Each class instance (object) has an independent RNG for all randomization methods in the class. When an object is created using **new**, its RNG is seeded with the next random value from the thread that creates the object. When a class object is created by a static declaration

initializer, there is no active thread; thus, the RNG of the created object is seeded with the next random value of the initialization RNG of the module instance, interface instance, program instance, or package in which the declaration occurred.

Object stability is guaranteed as long as object and thread creation and random number generation are done in the same order as before. In order to maintain random number stability, new objects, threads, and random numbers can be created after existing objects are created.

— *Manual seeding.* All noninitialization RNGs can be manually seeded. Combined with hierarchical seeding, this facility allows users to define the operation of a subsystem (hierarchy subtree) completely with a single seed at the root thread of the subsystem.

13.13.2 Thread stability

Random values returned from the `$urandom` system call are independent of thread execution order. For example:

```
integer x, y, z;
fork         //set a seed at the start of a thread
   begin process::self.srandom(100); x = $urandom; end
            //set a seed during a thread
   begin y = $urandom; process::self.srandom(200); end
            // draw 2 values from the thread RNG
   begin z = $urandom + $urandom ; end
join
```

The above program fragment illustrates several properties:

— *Thread locality.* The values returned for x, y, and z are independent of the order of thread execution. This is an important property because it allows development of subsystems that are independent, controllable, and predictable.

— *Hierarchical seeding.* When a thread is created, its random state is initialized using the next random value from the parent thread as a seed. The three forked threads are all seeded from the parent thread.

Each thread is seeded with a unique value, determined solely by its parent. The root of a thread execution subtree determines the random seeding of its children. This allows entire subtrees to be moved and preserves their behavior by manually seeding their root thread.

13.13.3 Object stability

The `randomize()` method built into every class exhibits object stability. This is the property that calls to `randomize()` in one instance are independent of calls to `randomize()` in other instances and are independent of calls to other randomize functions.

For example:

```
class Foo;
   rand integer x;
endclass

class Bar;
   rand integer y;
endclass

initial begin
   Foo foo = new();
```

```
        Bar bar = new();
        integer z;
        void'(foo.randomize());
        // z = $random;
        void'(bar.randomize());
    end
```

— The values returned for `foo.x` and `bar.y` are independent of each other.

— The calls to `randomize()` are independent of the `$random` system call. If one uncomments the line `z = $random` above, there is no change in the values assigned to `foo.x` and `bar.y`.

— Each instance has a unique source of random values that can be seeded independently. That random seed is taken from the parent thread when the instance is created.

— Objects can be seeded at any time using the `srandom()` method.

```
        class Foo;
            function new (integer seed);
                //set a new seed for this instance
                this.srandom(seed);
            endfunction
        endclass
```

Once an object is created, there is no guarantee that the creating thread can change the object's random state before another thread accesses the object. Therefore, it is best that objects self-seed within their **new** method rather than externally.

An object's seed can be set from any thread. However, a thread's seed can only be set from within the thread itself.

13.14 Manually seeding randomize

Each object maintains its own internal RNG, which is used exclusively by its `randomize()` method. This allows objects to be randomized independent of each other and of calls to other system randomization functions. When an object is created, its RNG is seeded using the next value from the RNG of the thread that creates the object. This process is called *hierarchical object seeding*.

Sometimes it is desirable to manually seed an object's RNG using the `srandom()` method. This can be done either in a class method or external to the class definition:

An example of seeding the RNG internally, as a class method, is as follows:

```
    class Packet;
        rand bit[15:0] header;
        ...
        function new (int seed);
            this.srandom(seed);
            ...
        endfunction
    endclass
```

An example of seeding the RNG externally is as follows:

```
    Packet p = new(200); // Create p with seed 200.
    p.srandom(300);      // Re-seed p with seed 300.
```

Calling `srandom()` in an object's **new**() function assures the object's RNG is set with the new seed before any class member values are randomized.

13.15 Random weighted case—randcase

```
statement_item ::=                                              // from A.6.4
        ...
      | randcase_statement
randcase_statement ::=                                          // from A.6.7
      randcase randcase_item { randcase_item } endcase
randcase_item ::= expression : statement_or_null
```

Syntax 13-11—Randcase syntax (excerpt from Annex A)

The keyword **randcase** introduces a **case** statement that randomly selects one of its branches. The randcase item expressions are non-negative integral values that constitute the branch weights. An item's weight divided by the sum of all weights gives the probability of taking that branch. For example:

```
randcase
   3 : x = 1;
   1 : x = 2;
   4 : x = 3;
endcase
```

The sum of all weights is 8; therefore, the probability of taking the first branch is 0.375, the probability of taking the second is 0.125, and the probability of taking the third is 0.5.

If a branch specifies a zero weight, then that branch is not taken. If all randcase items specify zero weights, then no branch is taken and a warning can be issued.

The randcase weights can be arbitrary expressions, not just constants. For example:

```
byte a, b;

randcase
   a + b : x = 1;
   a - b : x = 2;
   a ^ ~b : x = 3;
   12'b800 : x = 4;
endcase
```

The precision of each weight expression is self-determined. The sum of the weights is computed using standard addition semantics (maximum precision of all weights), where each summand is unsigned. Each weight expression is evaluated at most once (implementations can cache identical expressions) in an unspecified order. In the example above, the first three weight expressions are computed using 8-bit precision, and the fourth expression is computed using 12-bit precision. The resulting weights are added as unsigned values using 12-bit precision. The weight selection then uses unsigned 12-bit comparison.

Each call to **randcase** retrieves one random number in the range of 0 to the sum of the weights. The weights are then selected in declaration order: smaller random numbers correspond to the first (top) weight statements.

Randcase statements exhibit thread stability. The random numbers are obtained from `$urandom_range()`; thus, random values drawn are independent of thread execution order. This can result in multiple calls to `$urandom_range()` to handle greater than 32 bits.

13.16 Random sequence generation—randsequence

Parser generators, such as yacc, use a BNF or similar notation to describe the grammar of the language to be parsed. The grammar is thus used to generate a program that is able to check whether a stream of tokens represents a syntactically correct utterance in that language. SystemVerilog's sequence generator reverses this process. It uses the grammar to randomly create a correct utterance (i.e., a stream of tokens) of the language described by the grammar. The random sequence generator is useful for randomly generating structured sequences of stimulus such as instructions or network traffic patterns.

The sequence generator uses a set of rules and productions within a **randsequence** block. The syntax of the **randsequence** block is as follows:

```
statement_item ::=                                                          // from A.6.4

        ...
      | randsequence_statement

randsequence_statement ::= randsequence ( [ production_ identifier ] )       // from A.6.12
          production { production }
      endsequence

production ::= [ function_data_type ] production_identifier [ ( tf_port_list ) ] : rs_rule { | rs_rule } ;

rs_rule ::= rs_production_list [ := weight_specification [ rs_code_block ] ]

rs_production_list ::=
        rs_prod { rs_prod }
      | rand join [ ( expression ) ] production_item  production_item { production_item }

weight_specification ::=
        integral_number
      | ps_identifier
      | ( expression )

rs_code_block ::= { { data_declaration } { statement_or_null } }

rs_prod ::=
        production_item
      | rs_code_block
      | rs_if_else
      | rs_repeat
      | rs_case

production_item ::= production_identifier [ ( list_of_arguments ) ]

rs_if_else ::= if ( expression ) production_item [ else production_item ]

rs_repeat ::= repeat ( expression ) production_item

rs_case ::= case ( expression ) rs_case_item { rs_case_item } endcase

rs_case_item ::=
        expression { , expression } : production_item ;
      | default [ : ] production_item ;
```

Syntax 13-12—Randsequence syntax (excerpt from Annex A)

A **randsequence** grammar is composed of one or more productions. Each production contains a name and a list of production items. Production items are further classified into terminals and nonterminals.

Nonterminals are defined in terms of terminals and other nonterminals. A terminal is an indivisible item that needs no further definition than its associated code block. Ultimately, every nonterminal is decomposed into its terminals. A production list contains a succession of production items, indicating that the items must be streamed in sequence. A single production can contain multiple production lists separated by the | symbol. Production lists separated by a | imply a set of choices, which the generator will make at random.

A simple example illustrates the basic concepts:

```
randsequence( main )
    main     : first second done ;
    first    : add | dec ;
    second   : pop | push ;
    done     : { $display("done"); } ;
    add      : { $display("add");  } ;
    dec      : { $display("dec");  } ;
    pop      : { $display("pop");  } ;
    push     : { $display("push"); } ;
endsequence
```

The production `main` is defined in terms of three nonterminals: `first`, `second`, and `done`. When `main` is chosen, it generates the sequence, `first`, `second`, and `done`. When the first production is generated, it is decomposed into its productions, which specify a random choice between `add` and `dec`. Similarly, the second production specifies a choice between `pop` and `push`. All other productions are terminals; they are completely specified by their code block, which in the example displays the production name. Thus, the grammar leads to the following possible outcomes:

```
add pop done
add push done
dec pop done
dec push done
```

When the **randsequence** statement is executed, it generates a grammar-driven stream of random productions. As each production is generated, the side effects of executing its associated code blocks produce the desired stimulus. In addition to the basic grammar, the sequence generator provides for random weights, interleaving, and other control mechanisms. Although the **randsequence** statement does not intrinsically create a loop, a recursive production will cause looping.

The **randsequence** statement creates an automatic scope. All production identifiers are local to the scope. In addition, each code block within the **randsequence** block creates an anonymous automatic scope. Hierarchical references to the variables declared within the code blocks are not allowed. To declare a static variable, the static prefix must be used. The **randsequence** keyword can be followed by an optional production name (inside the parentheses) that designates the name of the top-level production. If unspecified, the first production becomes the top-level production.

13.16.1 Random production weights

The probability that a production list is generated can be changed by assigning weights to production lists. The probability that a particular production list is generated is proportional to its specified weight.

---

production ::= [function_data_type] production_identifier [**(** tf_port_list **)**] **:** rs_rule { **|** rs_rule } **;**

rs_rule ::= rs_production_list [**:=** weight_specification [rs_code_block]]

The `:=` operator assigns the weight specified by the *weight_specification* to its production list. A *weight_specification* must evaluate to an integral non-negative value. A weight is only meaningful when

assigned to alternative productions, that is, production list separated by a |. Weight expressions are evaluated when their enclosing production is selected, thus allowing weights to change dynamically. For example, the first production of the previous example can be rewritten as follows:

```
first : add := 3
| dec := (1 + 1)  // 2
;
```

This defines the production `first` in terms of two weighted production lists, `add` and `dec`. The production `add` will be generated with 60% probability, and the production `dec` will be generated with 40% probability.

If no weight is specified, a production shall use a weight of 1. If only some weights are specified, the unspecified weights shall use a weight of 1.

13.16.2 If...else production statements

A production can be made conditionally by means of an **if**...**else** production statement. The syntax of the **if**...**else** production statement is as follows:

rs_if_else ::= **if (** expression **)** production_item [**else** production_item]

The *expression* can be any expression that evaluates to a boolean value. If the expression evaluates to true, the production following the expression is generated; otherwise, the production following the optional else statement is generated. For example:

```
randsequence()
   ...
   PP_PO : if ( depth < 2 ) PUSH else POP ;
   PUSH  : { ++depth; do_push(); };
   POP   : { --depth; do_pop(); };
endsequence
```

This example defines the production PP_OP. If the variable depth is less than 2, then production PUSH is generated. Otherwise, production POP is generated. The variable depth is updated by the code blocks of both the PUSH and POP productions.

13.16.3 Case production statements

A production can be selected from a set of alternatives using a **case** production statement. The syntax of the **case** production statement is as follows:

rs_case ::= **case (** expression **)** rs_case_item { rs_case_item } **endcase**

rs_case_item ::=
      expression { **,** expression } **:** production_item **;**
   | **default** [**:**] production_item **;**

The case production statement is analogous to the procedural **case** statement except as noted below. The *case expression* is evaluated, and its value is compared against the value of each *case_item expression*, all of which are evaluated and compared in the order in which they are given. The production generated is the one associated with the first *case_item expression* matching the *case expression*. If no matching *case_item expression* is found, then the production associated with the optional default item is generated, or nothing if there is no default item. Multiple default statements in one case production statement shall be illegal. The *case_item expression*s separated by commas allow multiple expressions to share the production. For example:

```
randsequence()
   SELECT : case ( device & 7 )
      0        : NETWORK ;
      1, 2     : DISK ;
      default  : MEMORY ;
   endcase ;
   ...
endsequence
```

This example defines the production SELECT with a **case** statement. The case expression (device & 7) is evaluated and compared against the two case_item expressions. If the expression matches 0, the production NETWORK is generated; and if it matches 1 or 2, the production DISK is generated. Otherwise, the production MEMORY is generated.

13.16.4 Repeat production statements

The **repeat** production statement is used to iterate over a production a specified number of times. The syntax of the **repeat** production statement is as follows:

rs_repeat ::= **repeat (** expression **)** production_item

The **repeat** expression must evaluate to a non-negative integral value. That value specifies the number of times that the corresponding production is generated. For example:

```
randsequence()
   ...
   PUSH_OPER : repeat( $urandom_range( 2, 6 ) ) PUSH ;
   PUSH      : ...
endsequence
```

In this example, the PUSH_OPER production specifies that the PUSH production be repeated a random number of times (between 2 and 6) depending on the value returned by $urandom_range().

The **repeat** production statement itself cannot be terminated prematurely. A **break** statement will terminate the entire **randsequence** block (see 13.16.6).

13.16.5 Interleaving productions—rand join

The **rand join** production control is used to randomly interleave two or more production sequences while maintaining the relative order of each sequence. The syntax of the **rand join** production control is as follows:

rs_production_list ::=
      rs_prod { rs_prod }
   | **rand join** [**(** expression **)**] production_item production_item { production_item }

For example:

```
randsequence( TOP )
   TOP : rand join S1 S2 ;
   S1  : A B ;
   S2  : C D ;
endsequence
```

The generator will randomly produce the following sequences:

```
A  B  C  D
A  C  B  D
A  C  D  B
C  D  A  B
C  A  B  D
C  A  D  B
```

The optional expression following the **rand join** keywords must be a real number in the range of 0.0 to 1.0. The value of this expression represents the degree to which the length of the sequences to be interleaved affects the probability of selecting a sequence. A sequence's length is the number of productions not yet interleaved at a given time. If the expression is 0.0, the shortest sequences are given higher priority. If the expression is 1.0, the longest sequences are given priority. For instance, using the previous example,

```
TOP : rand join (0.0) S1 S2 ;
```

gives higher priority to the sequences: A B C D   C D A B, and

```
TOP : rand join (1.0) S1 S2 ;
```

gives higher priority to the sequences: A C B D   A C D B   C A B D   C A D B.

If unspecified, the generator used the default value of 0.5, which does not prioritize any sequence length.

At each step, the generator interleaves nonterminal symbols to depth of 1.

13.16.6 Aborting productions—break and return

Two procedural statements can be used to terminate a production prematurely: **break** and **return**. These two statements can appear in any code block; they differ in what they consider the scope from which to exit.

The **break** statement terminates the sequence generation. When a **break** statement is executed from within a production code block, it forces a jump out of the **randsequence** block. For example:

```
randsequence()
   WRITE   : SETUP DATA ;
   SETUP  : { if( fifo_length >= max_length ) break; } COMMAND ;
   DATA   : ...
endsequence
next_statement : ...
```

When the example above executes the **break** statement within the SETUP production, the COMMAND production is not generated, and execution continues on the line labeled next_statement. Use of the **break** statement within a loop statement behaves as defined in [10.6](#). Thus, the **break** statement terminates the smallest enclosing looping statement; otherwise, it terminates the **randsequence** block.

The **return** statement aborts the generation of the current production. When a **return** statement is executed from within a production code block, the current production is aborted. Sequence generation continues with the next production following the aborted production. For example:

```
randsequence()
   TOP : P1 P2 ;
   P1  : A B C ;
   P2  : A { if( flag == 1 ) return; } B C ;
   A   : { $display( "A" ); } ;
```

```
    B    : { if( flag == 2 ) return; $display( "B" ); } ;
    C    : { $display( "C" ); } ;
  endsequence
```

Depending on the value of variable `flag`, the example above displays the following:

```
  flag == 0  ==>  A B C A B C
  flag == 1  ==>  A B C A
  flag == 2  ==>  A C A C
```

When `flag == 1`, production `P2` is aborted in the middle, after generating `A`. When `flag == 2`, production `B` is aborted twice (once as part of `P1` and once as part of `P2`); however, each time, generation continues with the next production, `C`.

13.16.7 Value passing between productions

Data can be passed down to a production about to be generated, and generated productions can return data to the nonterminals that triggered their generation. Passing data to a production is similar to a task call and uses the same syntax. Returning data from a production requires that a type be declared for the production, which uses syntax similar to a function declaration.

Productions that accept data include a formal argument list. The syntax for declaring the arguments to a production is similar to a task prototype; the syntax for passing data to the production is the same as a task call.

---

production ::= [function_data_type] production_identifier [**(** tf_port_list **)**] **:** rs_rule { **|** rs_rule } **;**

production_item ::= production_identifier [**(** list_of_arguments **)**]

For example, the first example above could be written as follows:

```
  randsequence( main )
    main                    : first second gen ;
    first                   : add | dec ;
    second                  : pop | push ;
    add                     : gen("add") ;
    dec                     : gen("dec") ;
    pop                     : gen("pop") ;
    push                    : gen("push") ;
    gen( string s = "done" )  : { $display( s ); } ;
  endsequence
```

In this example, the production `gen` accepts a string argument whose default is `"done"`. Five other productions generate this production, each with a different argument (the one in `main` uses the default).

A production creates a scope, which encompasses all its rules and code blocks. Thus, arguments passed down to a production are available throughout the production.

Productions that return data require a type declaration. The optional return type precedes the production. Productions that do not specify a return type shall assume a void return type.

A value is returned from a production by using the **return** with an expression. When the **return** statement is used with a production that returns a value, it must specify an expression of the correct type, just like non-void functions. The **return** statement assigns the given expression to the corresponding production. The return value can be read in the code blocks of the production that triggered the generation of the production returning a value. Within these code blocks, return values are accessed using the production name plus an

optional indexing expression. Within each production, a variable of the same name is implicitly declared for each production that returns a value.

If the same production appears multiple times, then a one-dimensional array that starts at 1 is implicitly declared. For example:

```
randsequence( bin_op )
   void bin_op      : value operator value  // void type is optional
                      { $display( "%s %b %b", operator, value[1], value[2] );
}
                   ;
   bit [7:0] value  : { return $urandom } ;
   string operator  : add := 5 { return "+" ; }
                            | dec  := 2 { return "-" ; }
                            | mult := 1 { return "*" ; }
                        ;
   endsequence
```

In the example above, the `operator` and `value` productions return a string and an 8-bit value, respectively. The production `bin_op` includes these two value-returning productions. Therefore, the code block associated with production `bin_op` has access to the following implicit variable declarations:

```
bit [7:0] value [1:2];
string operator;
```

Accessing these implicit variables yields the values returned from the corresponding productions. When executed, the example above displays a simple three-item random sequence: an operator followed by two 8-bit values. The operators +, -, and * are chosen with a distribution of 5/8, 2/8, and 1/8, respectively.

Only the return values of productions already generated (i.e., to the left of the code block accessing them) can be retrieved. Attempting to read the return value of a production that has not been generated results in an undefined value. For example:

```
X : A {int y = B;} B ;                      // invalid use of B
X : A {int y = A[2];} B A ;                 // invalid use of A[2]
X : A {int y = A;} B {int j = A + B;} ;   // valid
```

The sequences produced by **randsequence** can be driven directly into a system, as a side effect of production generation, or the entire sequence can be generated for future processing. For example, the following function generates and returns a queue of random numbers in the range given by its arguments. The first and last queue item correspond to the lower and upper bounds, respectively. Also, the size of the queue is randomly selected based on the production weights.

```
function int[$] GenQueue(int low, int high);
   int[$] q;

   randsequence()
      TOP      : BOUND(low) LIST BOUND(high) ;
      LIST     : LIST ITEM  := 8  { q = { q, ITEM }; }
                  | ITEM   := 2  { q = { q, ITEM }; }
                ;
      int ITEM : { return $urandom_range( low, high ); } ;

      BOUND(int b) : { q = { q, b }; } ;
   endsequence
   GenQueue = q;
endfunction
```

When the **randsequence** in function GenQueue executes, it generates the TOP production, which causes three productions to be generated: BOUND with argument low, LIST, and BOUND with argument high. The BOUND production simply appends its argument to the queue. The LIST production consists of a weighted LIST ITEM production and an ITEM production. The LIST ITEM production is generated with 80% probability, which causes the LIST production to be generated recursively, thereby postponing the generation of the ITEM production. The selection between LIST ITEM and ITEM is repeated until the ITEM production is selected, which terminates the LIST production. Each time the ITEM production is generated, it produces a random number in the indicated range, which is later appended to the queue.

The following example uses a **randsequence** block to produce random traffic for a DSL packet network:

```
class DSL; ... endclass    // class that creates valid DSL packets

  randsequence (STREAM)
    STREAM : GAP DATA := 80
           | DATA      := 20 ;

    DATA   : PACKET(0)     := 94 { transmit( PACKET ); }
           | PACKET(1)     := 6  { transmit( PACKET ); } ;

    DSL PACKET (bit bad) : { DSL d = new;
                              if( bad ) d.crc ^= 23;     // mangle crc
                              return d;
                          );
    GAP: { ## {$urandom_range( 1, 20 )}; };
  endsequence
```

In this example, the traffic consists of a stream of (good and bad) data packets and gaps. The first production, STREAM, specifies that 80% of the time the traffic consists of a GAP followed by some DATA and 20% of the time it consists of just DATA (no GAP). The second production, DATA, specifies that 94% of all data packets are good packets and the remaining 6% are bad packets. The PACKET production implements the DSL packet creation; if the production argument is 1, then a bad packet is produced by mangling the crc of a valid DSL packet. Finally, the GAP production implements the transmission gaps by waiting a random number of cycles between 1 and 20.

14. Interprocess synchronization and communication

14.1 Introduction

NOTE—High-level and easy-to-use synchronization and communication mechanisms are essential to control the kinds of interactions that occur between dynamic processes used to model a complex system or a highly reactive testbench. Verilog provides basic synchronization mechanisms (i.e., `->` and `@`), but they are all limited to static objects. They are adequate for synchronization at the hardware level, but fall short of the needs of a highly dynamic, reactive testbench. At the system level, an essential limitation of Verilog is its inability to create dynamic events and communication channels that match the capability to create dynamic processes.

SystemVerilog adds a powerful and easy-to-use set of synchronization and communication mechanisms, all of which can be created and reclaimed dynamically. SystemVerilog adds a semaphore built-in class, which can be used for synchronization and mutual exclusion to shared resources, and a mailbox built-in class, which can be used as a communication channel between processes. SystemVerilog also enhances Verilog's named **event** data type to satisfy many of the system-level synchronization requirements.

Semaphores and mailboxes are built-in types; nonetheless, they are classes and can be used as base classes for deriving additional higher level classes. These built-in classes reside in the built-in `std` package (see 8.10.1); thus, they can be redefined by user code in any other scope.

14.2 Semaphores

Conceptually, a semaphore is a bucket. When a semaphore is allocated, a bucket that contains a fixed number of keys is created. Processes using semaphores must first procure a key from the bucket before they can continue to execute. If a specific process requires a key, only a fixed number of occurrences of that process can be in progress simultaneously. All others must wait until a sufficient number of keys is returned to the bucket. Semaphores are typically used for mutual exclusion, access control to shared resources, and basic synchronization.

An example of creating a semaphore is as follows:

```
semaphore smTx;
```

Semaphore is a built-in class that provides the following methods:
— Create a semaphore with a specified number of keys: **new**()
— Obtain one or more keys from the bucket: `get()`
— Return one or more keys into the bucket: `put()`
— Try to obtain one or more keys without blocking: `try_get()`

14.2.1 New()

Semaphores are created with the **new**() method.

The prototype for semaphore **new**() is as follows:

```
function new(int keyCount = 0 );
```

The `keyCount` specifies the number of keys initially allocated to the semaphore bucket. The number of keys in the bucket can increase beyond `keyCount` when more keys are put into the semaphore than are removed. The default value for `keyCount` is 0.

The **new**() function returns the semaphore handle or, if the semaphore cannot be created, **null**.

14.2.2 Put()

The semaphore `put()` method is used to return keys to a semaphore.

The prototype for `put()` is as follows:

```
task put(int keyCount = 1);
```

The `keyCount` specifies the number of keys being returned to the semaphore. The default is 1.

When the `semaphore.put()` task is called, the specified number of keys is returned to the semaphore. If a process has been suspended waiting for a key, that process shall execute if enough keys have been returned.

14.2.3 Get()

The semaphore `get()` method is used to procure a specified number of keys from a semaphore.

The prototype for `get()` is as follows:

```
task get(int keyCount = 1);
```

The `keyCount` specifies the required number of keys to obtain from the semaphore. The default is 1.

If the specified number of keys is available, the method returns and execution continues. If the specified number of keys is not available, the process blocks until the keys become available.

The semaphore waiting queue is first-in first-out (FIFO). This does not guarantee the order in which processes arrive at the queue, only that their arrival order shall be preserved by the semaphore.

14.2.4 Try_get()

The semaphore `try_get()` method is used to procure a specified number of keys from a semaphore, but without blocking.

The prototype for `try_get()` is as follows:

```
function int try_get(int keyCount = 1);
```

The `keyCount` specifies the required number of keys to obtain from the semaphore. The default is 1.

If the specified number of keys is available, the method returns a positive integer and execution continues. If the specified number of keys is not available, the method returns 0.

14.3 Mailboxes

A mailbox is a communication mechanism that allows messages to be exchanged between processes. Data can be sent to a mailbox by one process and retrieved by another.

Conceptually, mailboxes behave like real mailboxes. When a letter is delivered and put into the mailbox, one can retrieve the letter (and any data stored within). However, if the letter has not been delivered when one checks the mailbox, one must choose whether to wait for the letter or to retrieve the letter on a subsequent trip to the mailbox. Similarly, SystemVerilog's mailboxes provide processes to transfer and retrieve data in a controlled manner. Mailboxes are created as having either a bounded or unbounded queue size. A bounded mailbox becomes full when it contains the bounded number of messages. A process that attempts

to place a message into a full mailbox shall be suspended until enough room becomes available in the mailbox queue. Unbounded mailboxes never suspend a thread in a send operation.

An example of creating a mailbox is as follows:

```
mailbox mbxRcv;
```

Mailbox is a built-in class that provides the following methods:

— Create a mailbox: **new**()
— Place a message in a mailbox: put()
— Try to place a message in a mailbox without blocking: try_put()
— Retrieve a message from a mailbox: get() or peek()
— Try to retrieve a message from a mailbox without blocking: try_get() or try_peek()
— Retrieve the number of messages in the mailbox: num()

14.3.1 New()

Mailboxes are created with the **new**() method.

The prototype for mailbox **new**() is as follows:

```
function new(int bound = 0);
```

The **new**() function returns the mailbox handle or, if the mailbox cannot be created, **null**. If the bound argument is 0, then the mailbox is unbounded (the default) and a put() operation shall never block. If bound is nonzero, it represents the size of the mailbox queue.

The bound must be positive. Negative bounds are illegal and can result in indeterminate behavior, but implementations can issue a warning.

14.3.2 Num()

The number of messages in a mailbox can be obtained via the num() method.

The prototype for num() is as follows:

```
function int num();
```

The num() method returns the number of messages currently in the mailbox.

The returned value should be used with care because it is valid only until the next get() or put() is executed on the mailbox. These mailbox operations can be from different processes from the one executing the num() method. Therefore, the validity of the returned value shall depend on the time that the other methods start and finish.

14.3.3 Put()

The put() method places a message in a mailbox.

The prototype for put() is as follows:

```
task put( singular message);
```

The `message` is any singular expression, including object handles.

The `put()` method stores a message in the mailbox in strict FIFO order. If the mailbox was created with a bounded queue, the process shall be suspended until there is enough room in the queue.

14.3.4 Try_put()

The `try_put()` method attempts to place a message in a mailbox.

The prototype for `try_put()` is as follows:

```
function int try_put( singular message);
```

The `message` is any singular expression, including object handles.

The `try_put()` method stores a message in the mailbox in strict FIFO order. This method is meaningful only for bounded mailboxes. If the mailbox is not full, then the specified message is placed in the mailbox, and the function returns a positive integer. If the mailbox is full, the method returns 0.

14.3.5 Get()

The `get()` method retrieves a message from a mailbox.

The prototype for `get()` is as follows:

```
task get( ref singular message );
```

The `message` can be any singular expression, and it must be a valid left-hand expression.

The `get()` method retrieves one message from the mailbox, that is, removes one message from the mailbox queue. If the mailbox is empty, then the current process blocks until a message is placed in the mailbox. If the type of the `message` variable is not equivalent to the type of the message in the mailbox, a run-time error is generated.

Nonparameterized mailboxes are typeless, that is, a single mailbox can send and receive different types of data. Thus, in addition to the data being sent (i.e., the message queue), a mailbox implementation must maintain the message data type placed by `put()`. This is required in order to enable the run-time type checking.

The mailbox waiting queue is FIFO. This does not guarantee the order in which processes arrive at the queue, only that their arrival order shall be preserved by the mailbox.

14.3.6 Try_get()

The `try_get()` method attempts to retrieves a message from a mailbox without blocking.

The prototype for `try_get()` is as follows:

```
function int try_get( ref singular message );
```

The `message` can be any singular expression, and it must be a valid left-hand expression.

The `try_get()` method tries to retrieve one message from the mailbox. If the mailbox is empty, then the method returns 0. If the type of the `message` variable is not equivalent to the type of the message in the

mailbox, the method returns a negative integer. If a message is available and the message type is equivalent to the type of the `message` variable, the message is retrieved, and the method returns a positive integer.

14.3.7 Peek()

The `peek()` method copies a message from a mailbox without removing the message from the queue.

The prototype for `peek()` is as follows:

```
task peek( ref singular message );
```

The `message` can be any singular expression, and it must be a valid left-hand expression.

The `peek()` method copies one message from the mailbox without removing the message from the mailbox queue. If the mailbox is empty, then the current process blocks until a message is placed in the mailbox. If the type of the `message` variable is not equivalent to the type of the message in the mailbox, a run-time error is generated.

Calling the `peek()` method can also cause one message to unblock more than one process. As long as a message remains in the mailbox queue, any process blocked in either a `peek()` or `get()` operation shall become unblocked.

14.3.8 Try_peek()

The `try_peek()` method attempts to copy a message from a mailbox without blocking.

The prototype for `try_peek()` is as follows:

```
function int try_peek( ref singular message );
```

The `message` can be any singular expression, and it must be a valid left-hand expression.

The `try_peek()` method tries to copy one message from the mailbox without removing the message from the mailbox queue. If the mailbox is empty, then the method returns 0. If the type of the `message` variable is not equivalent to the type of the message in the mailbox, the method returns a negative integer. If a message is available and its type is equivalent to the type of the message variable, the message is copied, and the method returns a positive integer.

14.4 Parameterized mailboxes

The default mailbox is typeless, that is, a single mailbox can send and receive any type of data. This is a very powerful mechanism that, unfortunately, can also result in run-time errors due to type mismatches (types not equivalent) between a message and the type of the variable used to retrieve the message. Frequently, a mailbox is used to transfer a particular message type, and, in that case, it is useful to detect type mismatches at compile time.

Parameterized mailboxes use the same parameter mechanism as parameterized classes (see 7.23), modules, and interfaces:

```
mailbox #(type = dynamic_type)
```

where `dynamic_type` represents a special type that enables run-time type checking (the default).

A parameterized mailbox of a specific type is declared by specifying the type:

```
typedef mailbox #(string) s_mbox;

s_mbox sm = new;
string s;

sm.put( "hello" );
...
sm.get( s );   // s <- "hello"
```

Parameterized mailboxes provide all the same standard methods as dynamic mailboxes: `num()`, **`new()`**, `get()`, `peek()`, `put()`, `try_get()`, `try_peek()`, `try_put()`.

The only difference between a generic (dynamic) mailbox and a parameterized mailbox is that for a parameterized mailbox, the compiler ensures that the calls to `put`, `try_put`, `peek`, `try_peek`, `get`, and `try_get` methods use argument types equivalent to the mailbox type so that all type mismatches are caught by the compiler and not at run time.

14.5 Event

In Verilog, named events are static objects that can be triggered via the `->` operator, and processes can wait for an event to be triggered via the `@` operator. SystemVerilog events support the same basic operations, but enhance Verilog events in several ways. The most salient enhancement is that the triggered state of Verilog named events has no duration, whereas in SystemVerilog this state persists throughout the time step in which the event triggered. Also, SystemVerilog events act as handles to synchronization queues. Thus, they can be passed as arguments to tasks, and they can be assigned to one another or compared.

Existing Verilog event operations (`@` and `->`) are backward compatible and continue to work the same way when used in the static Verilog context. The additional functionality described below works with all events in either the static or dynamic context.

A SystemVerilog event provides a handle to an underlying synchronization object. When a process waits for an event to be triggered, the process is put on a queue maintained within the synchronization object. Processes can wait for a SystemVerilog event to be triggered either via the `@` operator or by the use of the `wait()` construct to examine their triggered state. Events are triggered using the `->` or the `->>` operator.

```
event_trigger ::=                                                        // from A.6.5
      -> hierarchical_event_identifier ;
    | ->> [ delay_or_event_control ] hierarchical_event_identifier ;
```

Syntax 14-1—Event trigger syntax (excerpt from Annex A)

The syntax to declare named events is discussed in 4.8.

14.5.1 Triggering an event

Named events are triggered via the `->` operator.

Triggering an event unblocks all processes currently waiting on that event. When triggered, named events behave like a one shot, i.e., the trigger state itself is not observable, only its effect. This is similar to the way in which an edge can trigger a flip-flop, but the state of the edge cannot be ascertained, i.e., `if (`**`posedge`** `clock)` is illegal.

14.5.2 Nonblocking event trigger

Nonblocking events are triggered using the `->>` operator.

The effect of the `->>` operator is that the statement executes without blocking and it creates a nonblocking assign update event in the time in which the delay control expires or the event control occurs. The effect of this update event shall be to trigger the referenced event in the nonblocking assignment region of the simulation cycle.

14.5.3 Waiting for an event

The basic mechanism to wait for an event to be triggered is via the event control operator, `@`.

```
@ hierarchical_event_identifier;
```

The `@` operator blocks the calling process until the given event is triggered.

For a trigger to unblock a process waiting on an event, the waiting process must execute the `@` statement before the triggering process executes the trigger operator, `->`. If the trigger executes first, then the waiting process remains blocked.

14.5.4 Persistent trigger: triggered property

SystemVerilog can distinguish the event trigger itself, which is instantaneous, from the event's triggered state, which persists throughout the time step (i.e., until simulation time advances). The `triggered` event property allows users to examine this state.

The `triggered` property is invoked using a method-like syntax:

```
hierarchical_event_identifier.triggered
```

The `triggered` event property evaluates to true if the given event has been triggered in the current time step and false otherwise. If `event_identifier` is **null**, then the `triggered` event property evaluates to false.

The `triggered` event property is most useful when used in the context of a **wait** construct:

```
wait ( hierarchical_event_identifier.triggered )
```

Using this mechanism, an event trigger shall unblock the waiting process whether the **wait** executes before or at the same simulation time as the trigger operation. The `triggered` event property, thus, helps eliminate a common race condition that occurs when both the trigger and the **wait** happen at the same time. A process that blocks waiting for an event might or might not unblock, depending on the execution order of the waiting and triggering processes. However, a process that waits on the triggered state always unblocks, regardless of the order of execution of the wait and trigger operations.

Example:

```
event done, blast;        // declare two new events
event done_too = done;    // declare done_too as alias to done

task trigger( event ev );
    -> ev;
endtask
```

```
    ...

    fork
       @ done_too;            // wait for done through done_too
       #1 trigger( done );    // trigger done through task trigger
    join

    fork
       -> blast;
       wait ( blast.triggered );
    join
```

The first fork in the example shows how two event identifiers, `done` and `done_too`, refer to the same synchronization object and also how an event can be passed to a generic task that triggers the event. In the example, one process waits for the event via `done_too`, while the actual triggering is done via the `trigger` task that is passed `done` as an argument.

In the second fork, one process can trigger the event `blast` before the other process (if the processes in the **fork**…**join** execute in source order) has a chance to execute, and wait for the event. Nonetheless, the second process unblocks and the fork terminates. This is because the process waits for the event's triggered state, which remains in its triggered state for the duration of the time step.

14.6 Event sequencing: wait_order()

The **wait_order** construct suspends the calling process until all of the specified events are triggered in the given order (left to right) or any of the untriggered events are triggered out of order and thus causes the operation to fail.

The syntax for the **wait_order** construct is as follows:

---

wait_statement ::=   *// from A.6.5*

    ...
    | **wait_order (** hierarchical_identifier { **,** hierarchical_identifier } **)** action_block
action_block ::=
      statement _or_null
    | [statement] **else** statement

Syntax 14-2—Wait_order event sequencing syntax (excerpt from Annex A)

For **wait_order** to succeed, at any point in the sequence, the subsequent events, which must all be untriggered at this point or the sequence would have already failed, must be triggered in the prescribed order. Preceding events are not limited to occur only once. In other words, once an event occurs in the prescribed order, it can be triggered again without causing the construct to fail.

Only the first event in the list can wait for the persistent `triggered` property.

The action taken when the construct fails depends on whether the optional phrase **else** statement (the fail statement) is specified. If it is specified, then the given statement is executed upon failure of the construct. If the fail statement is not specified, a failure generates a run-time error.

For example:

```
    wait_order( a, b, c);
```

suspends the current process until events a, b, and c trigger in the order a -> b -> c. If the events trigger out of order, a run-time error is generated.

Example:

```
wait_order( a, b, c ) else $display( "Error: events out of order" );
```

In this example, the fail statement specifies that, upon failure of the construct, a user message be displayed, but without an error being generated.

Example:

```
bit success;
wait_order( a, b, c ) success = 1; else success = 0;
```

In this example, the completion status is stored in the variable success, without an error being generated.

14.7 Event variables

An event is a unique data type with several important properties. Unlike Verilog, SystemVerilog events can be assigned to one another. When one event is assigned to another, the synchronization queue of the source event is shared by both the source and the destination event. In this sense, events act as full-fledged variables and not merely as labels.

14.7.1 Merging events

When one event variable is assigned to another, the two become merged. Thus, executing -> on either event variable affects processes waiting on either event variable.

For example:

```
event a, b, c;
a = b;
-> c;
-> a;    // also triggers b
-> b;    // also triggers a
a = c;
b = a;
-> a;    // also triggers b and c
-> b;    // also triggers a and c
-> c;    // also triggers a and b
```

When events are merged, the assignment only affects the execution of subsequent event control or wait operations. If a process is blocked waiting for event1 when another event is assigned to event1, the currently waiting process shall never unblock. For example:

```
fork
   T1: forever @ E2;
   T2: forever @ E1;
   T3: begin
        E2 = E1;
        forever -> E2;
   end
join
```

This example forks off three concurrent processes. Each process starts at the same time. Thus, at the same time that processes `T1` and `T2` are blocked, process `T3` assigns event `E1` to `E2`. As a result, process `T1` shall never unblock because the event `E2` is now `E1`. To unblock both threads `T1` and `T2`, the merger of `E2` and `E1` must take place before the fork.

14.7.2 Reclaiming events

When an event variable is assigned the special **null** value, the association between the event variable and the underlying synchronization queue is broken. When no event variable is associated with an underlying synchronization queue, the resources of the queue itself become available for reuse.

Triggering a **null** event shall have no effect. The outcome of waiting on a **null** event is undefined, and implementations can issue a run-time warning.

For example:

```
event E1 = null;
@ E1;                   // undefined: might block forever or not at all
wait( E1.triggered );   // undefined
-> E1;                  // no effect
```

14.7.3 Events comparison

Event variables can be compared against other event variables or the special value **null**. Only the following operators are allowed for comparing event variables:
 — Equality (`==`) with another event or with **null**
 — Inequality (`!=`) with another event or with **null**
 — Case equality (`===`) with another event or with **null** (same semantics as `==`)
 — Case inequality (`!==`) with another event or with **null** (same semantics as `!=`)
 — Test for a boolean value that shall be 0 if the event is **null** and 1 otherwise

Example:

```
event E1, E2;
if ( E1 )   // same as if ( E1 != null )
   E1 = E2;
if ( E1 == E2 )
   $display( "E1 and E2 are the same event" );
```

15. Clocking blocks

15.1 Introduction

NOTE—In Verilog, the communication between blocks is specified using module ports. SystemVerilog adds the interface, a key construct that encapsulates the communication between blocks, thereby enabling users to easily change the level of abstraction at which the intermodule communication is to be modeled.

An interface can specify the signals or nets through which a testbench communicates with a device under test (DUT). However, an interface does not explicitly specify any timing disciplines, synchronization requirements, or clocking paradigms.

SystemVerilog adds the **clocking** block that identifies clock signals and captures the timing and synchronization requirements of the blocks being modeled. A **clocking** block assembles signals that are synchronous to a particular clock and makes their timing explicit. The **clocking** block is a key element in a cycle-based methodology, which enables users to write testbenches at a higher level of abstraction. Rather than focusing on signals and transitions in time, the test can be defined in terms of cycles and transactions. Depending on the environment, a testbench can contain one or more **clocking** blocks, each containing its own clock plus an arbitrary number of signals.

The **clocking** block separates the timing and synchronization details from the structural, functional, and procedural elements of a testbench. Thus, the timing for sampling and driving **clocking** block signals is implicit and relative to the **clocking** block's clock. This enables a set of key operations to be written very succinctly, without explicitly using clocks or specifying timing. These operations are as follows:

— Synchronous events
— Input sampling
— Synchronous drives

15.2 Clocking block declaration

The syntax for the **clocking** block is as follows:

```
clocking_declaration ::=                                              // from A.6.11
      [ default ] clocking [ clocking_identifier ] clocking_event ;
          { clocking_item }
      endclocking [ : clocking_identifier ]
clocking_event ::=
      @ identifier
    | @ ( event_expression )
clocking_item ::=
      default default_skew ;
    | clocking_direction list_of_clocking_decl_assign ;
    | { attribute_instance } concurrent_assertion_item_declaration
default_skew ::=
      input clocking_skew
    | output clocking_skew
    | input clocking_skew output clocking_skew
clocking_direction ::=
      input [ clocking_skew ]
    | output [ clocking_skew ]
    | input [ clocking_skew ] output [ clocking_skew ]
    | inout
```

Syntax 15-1—Clocking block syntax (excerpt from Annex A)

list_of_clocking_decl_assign ::= clocking_decl_assign { , clocking_decl_assign }

clocking_decl_assign ::= signal_identifier [**=** expression]

clocking_skew ::=
      edge_identifier [delay_control]
    | delay_control

edge_identifier ::= **posedge** | **negedge**   *// from A.7.4*

delay_control ::=   *// from A.6.5*
      # delay_value
    | **# (** mintypmax_expression **)**

Syntax 15-1—Clocking block syntax (excerpt from Annex A) (continued)

The *delay_control* must be either a time literal or a constant expression that evaluates to a positive integer value.

The *clocking_identifier* specifies the name of the **clocking** block being declared.

The *signal_identfier* identifies a signal in the scope enclosing the **clocking** block declaration and declares the name of a signal in the **clocking** block. Unless a *hierarchical_expression* is used, both the signal and the *clocking_item* names shall be the same.

The *clocking_event* designates a particular event to act as the clock for the **clocking** block. Typically, this expression is either the **posedge** or **negedge** of a clocking signal. The timing of all the other signals specified in a given **clocking** block is governed by the clocking event. All **input** or **inout** signals specified in the **clocking** block are sampled when the corresponding clock event occurs. Likewise, all **output** or **inout** signals in the **clocking** block are driven when the corresponding clock event occurs. Bidirectional signals (**inout**) are sampled as well as driven. An **output** signal cannot be read, and an **input** signal cannot be driven.

The *clocking_skew* determines how many time units away from the clock event a signal is to be sampled or driven. Input skews are implicitly negative, that is, they always refer to a time before the clock, whereas output skews always refer to a time after the clock (see 15.3). When the clocking event specifies a simple edge, instead of a number, the skew can be specified as the specific edge of the signal. A single skew can be specified for the entire block by using a **default** clocking item.

```
clocking ck1 @(posedge clk);
   default input #1step output negedge; // legal
   // outputs driven on the negedge clk
   input ... ;
   output ... ;
endclocking

clocking ck2 @(clk); // no edge specified!
   default input #1step output negedge; // legal
   input ... ;
   output ... ;
endclocking
```

The *hierarchical_identifier* specifies that, instead of a local port, the signal to be associated with the **clocking** block is specified by its hierarchical name (cross-module reference).

Example:

```
clocking bus @(posedge clock1);
    default input #10ns output #2ns;
    input data, ready, enable = top.mem1.enable;
    output negedge ack;
    input #1step addr;
endclocking
```

In the above example, the first line declares a `clocking` block called `bus` that is to be clocked on the positive edge of the signal `clock1`. The second line specifies that by default all signals in the `clocking` block shall use a `10ns` input skew and a `2ns` output skew. The next line adds three input signals to the `clocking` block: `data`, `ready`, and `enable`; the last signal refers to the hierarchical signal `top.mem1.enable`. The fourth line adds the signal `ack` to the `clocking` block and overrides the default output skew so that `ack` is driven on the negative edge of the clock. The last line adds the signal `addr` and overrides the default input skew so that `addr` is sampled one step before the positive edge of the clock.

Unless otherwise specified, the default `input` skew is `1step` and the default `output` skew is `0`. A step is a special time unit whose value is defined in 19.10. A `1step` input skew allows input signals to sample their steady-state values in the time step immediately before the clock event (i.e., in the preceding Postponed region). Unlike other time units, which represent physical units, a step cannot be used to set or modify either the precision or the time unit.

15.3 Input and output skews

Input (or inout) signals are sampled at the designated clock event. If an input skew is specified, then the signal is sampled at *skew* time units *before* the clock event. Similarly, output (or inout) signals are driven *skew* simulation time units *after* the corresponding clock event. Figure 15-1 shows the basic sample and drive timing for a positive edge clock.
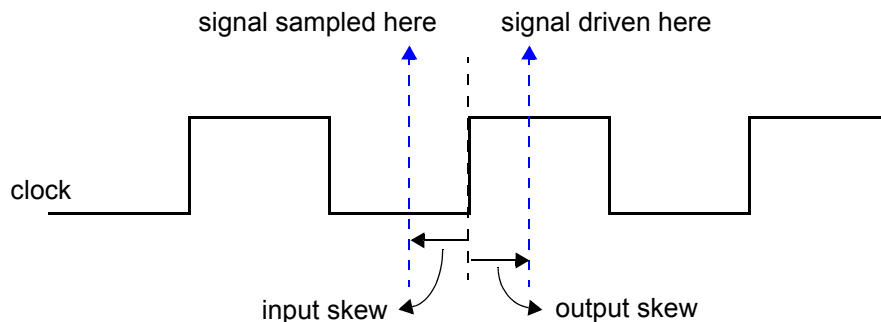
**Figure 15-1—Sample and drive times including skew
with respect to the positive edge of the clock**

A skew must be a constant expression and can be specified as a parameter. If the skew does not specify a time unit, the current time unit is used. If a number is used, the skew is interpreted using the timescale of the current scope.

```
clocking dram @(clk);
    input #1ps address;
    input #5 output #6 data;
endclocking
```

An input skew of `1step` indicates that the signal is to be sampled at the end of the previous time step. In other words, the value sampled is always the signal's last value immediately before the corresponding clock edge.

NOTE—A **clocking** block does not eliminate potential races when an event control outside of a program block is sensitive to the same clock as the **clocking** block and a statement after the event control attempts to read a member of the **clocking** block. The race is between reading the old sampled value and the new sampled value.

Inputs with explicit #0 skew shall be sampled at the same time as their corresponding clocking event, but to avoid races, they are sampled in the Observed region. Likewise, **clocking** block outputs with no skew (or explicit #0 skew) shall be driven at the same time as their specified clocking event, as nonblocking assignments (in the NBA region).

Skews are declarative constructs; thus, they are semantically very different from the syntactically similar procedural delay statement. In particular, an explicit #0 skew does not suspend any process, nor does it execute or sample values in the Inactive region.

15.4 Hierarchical expressions

Any signal in a **clocking** block can be associated with an arbitrary hierarchical expression. As described in 15.2, a hierarchical expression is introduced by appending an equal sign (=) followed by the hierarchical expression:

```
clocking cd1 @(posedge phi1);
      input #1step state = top.cpu.state;
endclocking
```

However, hierarchical expressions are not limited to simple names or signals in other scopes. They can be used to declare slices and concatenations (or combinations thereof) of signals in other scopes or in the current scope.

```
clocking mem @(clock);
    input instruction = { opcode, regA, regB[3:1] };
endclocking
```

In a **clocking** block, any expression assigned to a signal in its declaration shall be an expression that would be legal in a port connection to a port of any of the directions specified in the declaration. For example, it would be illegal to assign an **inout** signal an expression in its declaration that would be illegal in a port connection to an **inout** port.

15.5 Signals in multiple clocking blocks

The same signals—clock, inputs, inouts, or outputs—can appear in more than one **clocking** block. When **clocking** blocks use the same clock (or clocking expression), they shall share the same synchronization event, in the same manner as several latches can be controlled by the same clock. Input semantics is described in 15.12, and output semantics is described in 15.14.

15.6 Clocking block scope and lifetime

A **clocking** block is both a declaration and an instance of that declaration. A separate instantiation step is not necessary. Instead, one copy is created for each instance of the block containing the declaration (like an **always** block). Once declared, the clocking signals are available via the **clocking** block name and the dot (**.**) operator:

```
       dom.sig  // signal sig in clocking dom
```

Multiple **clocking** blocks cannot be nested. They cannot be declared inside functions, tasks, or packages or outside all declarations in a compilation unit. A **clocking** block can only be declared inside a module, interface, or program (see Clause 16).

A **clocking** block has static lifetime and scope local to its enclosing module, interface, or program.

15.7 Multiple clocking blocks example

In this example, a simple test program includes two **clocking** blocks. The program construct used in this example is discussed in Clause 16.

```
program test(  input phi1, input [15:0] data, output logic write,
               input phi2, inout [8:1] cmd, input enable
            );
    reg [8:1] cmd_reg;

    clocking cd1 @(posedge phi1);
       input data;
       output write;
       input state = top.cpu.state;
    endclocking

    clocking cd2 @(posedge phi2);
       input #2 output #4ps cmd;
       input enable;
    endclocking

    initial begin
       // program begins here
    ...
       // user can access cd1.data , cd2.cmd , etc…
    end
    assign cmd = enable ? cmd_reg: 'x;
endprogram
```

The test program can be instantiated and connected to a DUT (cpu and mem).

```
module top;
    logic phi1, phi2;
    wire [8:1] cmd; // cannot be logic (two bidirectional drivers)
    logic [15:0] data;

    test main( phi1, data, write, phi2, cmd, enable );
    cpu cpu1( phi1, data, write );
    mem mem1( phi2, cmd, enable );
endmodule
```

15.8 Interfaces and clocking blocks

A **clocking** encapsulates a set of signals that share a common clock; therefore, specifying a **clocking** block using a SystemVerilog **interface** can significantly reduce the amount of code needed to connect the testbench. Furthermore, because the signal directions in the **clocking** block within the testbench are with respect to the testbench and not the design under test, a **modport** declaration can appropriately describe

either direction. A testbench program can be contained within a program, and its ports can be interfaces that correspond to the signals declared in each **clocking** block. The interface's wires shall have the same direction as specified in the **clocking** block when viewed from the testbench side (i.e., **modport** test) and reversed when viewed from the DUT (i.e., **modport** dut).

For example, the previous example could be rewritten using interfaces as follows:

```
interface bus_A (input clk);
      logic [15:0] data;
      logic write;
      modport test (input data, output write);
      modport dut (output data, input write);
endinterface

interface bus_B (input clk);
      logic [8:1] cmd;
      logic enable;
      modport test (input enable);
      modport dut (output enable);
endinterface


program test( bus_A.test a, bus_B.test b );

      clocking cd1 @(posedge a.clk);
         input a.data;
         output a.write;
         inout state = top.cpu.state;
      endclocking

      clocking cd2 @(posedge b.clk);
         input #2 output #4ps b.cmd;
         input b.enable;
      endclocking

   initial begin
      // program begins here
      ...
      // user can access cd1.a.data , cd2.b.cmd , etc…
   end
endprogram
```

The test module can be instantiated and connected as before:

```
module top;
   logic phi1, phi2;

   bus_A a(phi1);
   bus_B b(phi2);

   test main( a, b );
   cpu cpu1( a );
   mem mem1( b );
endmodule
```

Alternatively, in the program test above, the **clocking** block can be written using both interfaces and hierarchical expressions as follows:

```
clocking cd1 @(posedge a.clk);
    input data = a.data;
    output write = a.write;
    inout state = top.cpu.state;
endclocking

clocking cd2 @(posedge b.clk);
    input #2 output #4ps cmd = b.cmd;
    input enable = b.enable;
endclocking
```

This would allow using the shorter names (`cd1.data`, `cd2.cmd`, …) instead of the longer interface syntax (`cd1.a.data`, `cd2.b.cmd`, …).

15.9 Clocking block events

The clocking event of a **clocking** block is available directly by using the **clocking** block name, regardless of the actual clocking event used to declare the **clocking** block.

For example.

```
clocking dram @(posedge phi1);
    inout data;
    output negedge #1 address;
endclocking
```

The clocking event of the `dram` **clocking** block can be used to wait for that particular event:

```
@( dram );
```

The above statement is equivalent to @(**posedge** phi1).

15.10 Cycle delay:

The ## operator can be used to delay execution by a specified number of clocking events or clock cycles.

The syntax for the cycle delay statement is as follows:

---

procedural_timing_control_statement ::=   // from A.6.5
      procedural_timing_control statement_or_null

procedural_timing_control ::=

      ...
    | cycle_delay

cycle_delay ::=   // from A.6.11
      ## integral_number
    | ## identifier
    | ## (expression)

Syntax 15-2—Cycle delay syntax (excerpt from Annex A)

The *expression* can be any SystemVerilog expression that evaluates to a positive integer value.

What constitutes a cycle is determined by the default clocking in effect (see 15.11). If no default clocking has been specified for the current module, interface, or program, then the compiler shall issue an error.

Example:

```
## 5;            // wait 5 cycles (clocking events) using the default clocking

## (j + 1);      // wait j+1 cycles (clocking events) using the default clocking
```

15.11 Default clocking

One **clocking** can be specified as the default for all cycle delay operations within a given module, interface, or program.

The syntax for the default cycle specification statement is as follows:

---

module_or_generate_item_declaration ::=   // from A.1.4
   ...
   | **default clocking** clocking_identifier ;
clocking_declaration ::=   // from A.6.11
   [**default**] **clocking** [clocking_identifier] clocking_event **;**
     { clocking_item }
   endclocking [**:** clocking_identifier]

Syntax 15-3—Default clocking syntax (excerpt from Annex A)

The *clocking_identifier* must be the name of a **clocking** block.

Only one default clocking can be specified in a program, module, or interface. Specifying a default clocking more than once in the same program or module shall result in a compiler error.

A default clocking is valid only within the scope containing the default clocking specification. This scope includes the module, interface, or program that contains the declaration as well as any nested modules or interfaces. It does not include instantiated modules or interfaces.

Example 1: Declaring a clocking as the default:

```
program test( input bit clk, input reg [15:0] data );
   default clocking bus @(posedge clk);
      inout data;
   endclocking

   initial begin
      ## 5;
      if ( bus.data == 10 )
         ## 1;
      else
         ...
   end
endprogram
```

Example 2: Assigning an existing clocking to be the default:

```
module processor ...
```

```
        clocking busA @(posedge clk1); ... endclocking
        clocking busB @(negedge clk2); ... endclocking
      module cpu( interface y );
         default clocking busA ;
         initial begin
            ## 5; // use busA => (posedge clk1)
             ...
         end
      endmodule
   endmodule
```

15.12 Input sampling

All **clocking** block inputs (input or inout) are sampled at the corresponding clocking event. If the input skew is not an explicit #0, then the value sampled corresponds to the signal value at the Postponed region of the time step skew time units prior to the clocking event (see Figure 15-1 in 15.3). If the input skew is an explicit #0, then the value sampled corresponds to the signal value in the Observed region.

When a signal appears in an expression, it is replaced by the signal's sampled value, that is, the value that was sampled at the last sampling point.

When the same signal is an input to multiple **clocking** blocks, the semantics is straightforward; each **clocking** block samples the corresponding signal with its own clocking event.

15.13 Synchronous events

Explicit synchronization is done via the event control operator, @, which allows a process to wait for a particular signal value change or a clocking event (see 15.9).

The syntax for the synchronization operator is given in 10.10.

The expression used with the event control can denote **clocking** block input (**input** or **inout**) or a slice thereof. Slices can include dynamic indices, which are evaluated once, when the @ expression executes.

These are some examples of synchronization statements:
— Wait for the next change of signal ack_1 of **clocking** block ram_bus

```
@(ram_bus.ack_l);
```

— Wait for the next clocking event in **clocking** block ram_bus

```
@(ram_bus);
```

— Wait for the positive edge of the signal ram_bus.enable

```
@(posedge ram_bus.enable);
```

— Wait for the falling edge of the specified 1-bit slice dom.sign[a]

```
@(negedge dom.sign[a]);
```

NOTE—The index a is evaluated at run time.
— Wait for either the next positive edge of dom.sig1 or the next change of dom.sig2, whichever happens first

```
@(posedge dom.sig1 or dom.sig2);
```

— Wait for the either the negative edge of `dom.sig1` or the positive edge of `dom.sig2`, whichever happens first

```
@(negedge dom.sig1 or posedge dom.sig2);
```

The values used by the synchronization event control are the synchronous values, that is, the values sampled at the corresponding clocking event.

15.14 Synchronous drives

The `clocking` block outputs (`output` or `inout`) are used to drive values onto their corresponding signals, but at a specified time. In other words, the corresponding signal changes value at the indicated clocking event as modified by the output skew.

The syntax to specify a synchronous drive is similar to an assignment:

statement ::= [block_identifier **:**] { attribute_instance } statement_item   *// from A.6.4*

statement_item ::=

    ...

    | clocking_drive **;**

clocking_drive ::=   *// from A.6.11*

    clockvar_expression **<=** [cycle_delay] expression

    | cycle_delay clockvar_expression **<=** expression

cycle_delay ::= **##** expression

clockvar ::= hierarchical_identifier

clockvar_expression ::= clockvar select

Syntax 15-4—Synchronous drive syntax (excerpt from Annex A)

The *clockvar_expression* is either a bit-select, slice, or the entire **clocking** block output whose corresponding signal is to be driven (concatenation is not allowed):

```
dom.sig            // entire clockvar

dom.sig[2]         // bit-select

dom.sig[8:2]       // slice
```

The *expression* (in the *clocking_drive* production) can be any valid expression that is assignment compatible with the type of the corresponding signal.

The *event_count* refers to the *expression* after the ## in the *cycle_delay* production and is an integral *expression* that optionally specifies the number of clocking events (i.e., cycles) that must pass before the statement executes. Specifying a nonzero `event_count` blocks the current process until the specified number of clocking events has elapsed; otherwise, the statement executes at the current time. The `event_count` uses syntax similar to the cycle delay operator (see 15.10); however, the synchronous drive uses the **clocking** block of the signal being driven and not the default clocking.

The second form of the synchronous drive uses the intra-assignment syntax. An intra-assignment `event_count` specification also delays execution of the assignment. In this case, the process does not block, and the right-hand expression is evaluated when the statement executes.

Examples:

```
bus.data[3:0] <= 4'h5;  // drive data in the NBA region of the current cycle

##1 bus.data <= 8'hz;   // wait 1 (bus) cycle and then drive data

##2; bus.data <= 2;     // wait 2 default clocking cycles, then drive data

bus.data <= ##2 r;      // remember the value of r and then drive
                        // data 2 (bus) cycles later
```

Regardless of when the drive statement executes (due to event_count delays), the driven value is assigned to the corresponding signal only at the time specified by the output skew.

15.14.1 Drives and nonblocking assignments

Synchronous signal drives are processed as nonblocking assignments.

A key feature of **inout clocking** block variables and synchronous drives is that a drive does not change the **clocking** block input. This is because reading the input always yields the last sampled value, and not the driven value.

15.14.2 Drive value resolution

When more than one synchronous drive is applied to the same **clocking** block output (or inout) at the same simulation time, the driven values are checked for conflicts. When conflicting drives are detected, a run-time error is issued, and each conflicting bit is driven to X (or 0 for a 2-state port).

For example:

```
clocking pe @(posedge clk);
   output nibble;   // four bit output
endclocking

pe.nibble <= 4'b0101;
pe.nibble <= 4'b0011;
```

The driven value of nibble is 4'b0xx1, regardless of whether nibble is a **reg** or a **wire**.

When the same variable is an output from multiple **clocking** blocks, the last drive determines the value of the variable. This allows a single module to model multirate devices, such as a DDR memory, using a different **clocking** block to model each active edge. For example:

```
reg j;

clocking pe @(posedge clk);
   output j;
endclocking

clocking ne @(negedge clk);
   output j;
endclocking
```

The variable j is an output to two **clocking** blocks using different clocking events (**posedge** versus **negedge**). When driven, the variable j shall take on the value most recently assigned by either **clocking** block.

The **clocking** block outputs driving a net (i.e., through different ports) cause the net to be driven to its resolved signal value. When a **clocking** block output corresponds to a wire, a driver for that wire is created that is updated as if by a continuous assignment from a register inside the **clocking** block that is updated as a nonblocking assignment.

16. Program block

16.1 Introduction

NOTE—The module is the basic building block in Verilog. Modules can contain hierarchies of other modules, wires, task and function declarations, and procedural statements within **always** and **initial** blocks. This construct works extremely well for the description of hardware. However, for the testbench, the emphasis is not in the hardware-level details such as wires, structural hierarchy, and interconnects, but in modeling the complete environment in which a design is verified. A lot of effort is spent getting the environment properly initialized and synchronized, avoiding races between the design and the testbench, automating the generation of input stimuli, and reusing existing models and other infrastructure.

The program block serves three basic purposes:

— It provides an entry point to the execution of testbenches.

— It creates a scope that encapsulates programwide data, tasks, and functions.

— It provides a syntactic context that specifies scheduling in the Reactive region.

The program construct serves as a clear separator between design and testbench, and, more importantly, it specifies specialized execution semantics in the Reactive region for all elements declared within the program. Together with **clocking** blocks, the program construct provides for race-free interaction between the design and the testbench and enables cycle- and transaction-level abstractions.

The abstraction and modeling constructs of SystemVerilog simplify the creation and maintenance of testbenches. The ability to instantiate and individually connect each program instance enables their use as generalized models.

16.2 The program construct

A typical program contains type and data declarations, subroutines, connections to the design, and one or more procedural code streams. The connection between design and testbench uses the same interconnect mechanism as used by SystemVerilog to specify port connections, including interfaces. The syntax for the program block is as follows:

```
program_nonansi_header ::=                                                    // from A.1.2
        { attribute_instance } program [ lifetime ] program_identifier
            [ parameter_port_list ] list_of_ports ;
program_ansi_header ::=
        {attribute_instance } program [ lifetime ] program_identifier
            [ parameter_port_list ] [ list_of_port_declarations ] ;
program_declaration ::=
        program_nonansi_header [ timeunits_declaration ] { program_item }
            endprogram [ : program_identifier ]
    | program_ansi_header [ timeunits_declaration ] { non_port_program_item }
            endprogram [ : program_identifier ]
    | { attribute_instance } program program_identifier ( .* ) ;
        [ timeunits_declaration ] { program_item }
     endprogram [ : program_identifier ]
    | extern program_nonansi_header
    | extern program_ansi_header
program_item ::=                                                              // from A.1.7
        port_declaration ;
    | non_port_program_item
```

Syntax 16-1—Program declaration syntax (excerpt from Annex A)

```
non_port_program_item ::=
        { attribute_instance } continuous_assign
      | { attribute_instance } module_or_generate_item_declaration
      | { attribute_instance } initial_construct
      | { attribute_instance } final_construct
      | { attribute_instance } concurrent_assertion_item
      | { attribute_instance } timeunits_declaration17
      | program_generate_item

program_generate_item37 ::=
        loop_generate_construct
      | conditional_generate_construct
      | generate_region

lifetime ::= static | automatic                                                    // from A.2.1.3

anonymous_program ::= program ; { anonymous_program_item } endprogram              // from A.1.10

anonymous_program_item ::=
        task_declaration
      | function_declaration
      | class_declaration
      | covergroup_declaration
      | class_constructor_declaration
      | ;
```

Syntax 16-1—Program declaration syntax (excerpt from Annex A) (continued)

For example:

```
program test (input clk, input [16:1] addr, inout [7:0] data);
   initial ...
endprogram
```

or

```
program test ( interface device_ifc );
   initial ...
endprogram
```

A more complete example is included in 15.7 and 15.8.

Although the **program** construct is new to SystemVerilog, its inclusion is a natural extension. The **program** construct can be considered a leaf module with special execution semantics. Once declared, a program block can be instantiated in the required hierarchical location (typically at the top level), and its ports can be connected in the same manner as any other module.

Program blocks can be nested within modules or interfaces. This allows multiple cooperating programs to share variables local to the scope. Nested programs with no ports or top-level programs that are not explicitly instantiated are implicitly instantiated once. Implicitly instantiated programs have the same instance and declaration name. For example:

```
module test(...)
   int shared; // variable shared by programs p1 and p1

   program p1;
      ...
```

```
        endprogram


        program p2;
           ...
        endprogram // p1 and p2 are implicitly instantiated once in module test


    endmodule
```

A program block can contain one or more **initial** or **final** blocks. It cannot contain **always** blocks, UDPs, modules, interfaces, or other programs.

Type and data declarations within the program are local to the program scope and have static lifetime. Variables declared within the scope of a program are called *program variables*. Program variables can only be assigned using blocking assignments. Nonprogram variables can only be assigned using nonblocking assignments. Using nonblocking assignments with program variables or blocking assignments with design (nonprogram) variables shall be an error. References to program variables from outside any program block shall be an error.

16.3 Eliminating testbench races

There are two major sources of nondeterminism in Verilog. The first one is that active events are processed in an arbitrary order. The second one is that statements without time control constructs in behavioral blocks do not execute as one event. However, from the testbench perspective, these effects are all unimportant details. The primary task of a testbench is to generate valid input stimulus for the design under test and to verify that the device operates correctly. Furthermore, testbenches that use cycle abstractions are only concerned with the stable or steady state of the system for both checking the current outputs and for computing stimuli for the next cycle. Formal tools also work in this fashion.

Statements within a program block that are sensitive to changes (e.g., update events) in design signals (declared in modules, not program blocks) are scheduled in the Reactive region. Consider a program block that contains the statement @(clk) S1; where clk is a design signal in some module. Every transition of signal clk will cause the statement S1 to be scheduled into the Reactive region. Likewise, **initial** blocks within program blocks are scheduled in the Reactive region; in contrast, **initial** blocks in modules are scheduled in the Active region. In addition, design signals driven from within the program must be assigned using nonblocking assignments and are updated in the NBA region. Thus, even signals driven with no delay are propagated into the design as one event. With this behavior, correct cycle semantics can be modeled without races, thereby making program-based testbenches compatible with clocked assertions and formal tools.

Because the program schedules events in the Reactive region, the **clocking** block construct is very useful to automatically sample the steady-state values of previous time steps or clock cycles. Programs that read design values exclusively through **clocking** blocks with #0 input skews are insensitive to read-write races. It is important to understand that simply sampling input signals (or setting nonzero skews on **clocking** block inputs) does not eliminate the potential for races. Proper input sampling only addresses a single **clocking** block. With multiple clocks, the arbitrary order in which overlapping or simultaneous clocks are processed is still a potential source for races. The program construct addresses this issue by scheduling its execution in the Reactive region, after all design events have been processed, including clocks driven by nonblocking assignments.

16.3.1 Zero-skew clocking block races

When a **clocking** block sets both input and output skews to #0 (see 15.3), then its inputs are sampled (in the Observed region) at the same time as its outputs are driven (in the NBA region). This type of explicit #0

delay processing is a common source of nondeterminism that can result in races. Nonetheless, even in this case, the program minimizes races by means of two mechanisms:

— First, by constraining program statements to be scheduled in the Reactive region, after all explicit #0 delay transitions have propagated through the design and the system has reached a quasi steady state.

— Second, by requiring design variables or nets to be modified only via nonblocking assignments.

These two mechanisms reduce the likelihood of a race; nonetheless, a race is still possible when skews are set to explicit #0.

16.4 Blocking tasks in cycle/event mode

Calling program tasks or functions from within design modules is illegal and shall result in an error. This is because the design must not be aware of the testbench. Programs are allowed to call tasks or functions in other programs or within design modules. Functions within design modules can be called from a program and require no special handling. However, blocking tasks (i.e., tasks that do not execute in 0 simulation time) within design modules that are called from a program do require explicit synchronization upon return from the tasks. In other words, when blocking tasks return to the program code, the program block execution is automatically postponed until the Reactive region. The copy out of the parameters happens when the task returns.

Calling blocking tasks in design modules from within programs requires careful consideration. Expressions evaluated by the task before blocking on the first timing control shall use the values after they have been updated by nonblocking assignments in the design module. In contrast, if the task is called from a module at the start of the time step (before nonblocking assignments are processed), then those same expressions shall use the values before they have been updated by nonblocking assignments.

```
module ...
   task T;
      S1: a = b;      // might execute before or after the Observe region
      #5;
      S2: b <= 1'b1; // always executes before the Observe region
   endtask
endmodule
```

If task `T`, above, is called from within a module, then the statement `S1` can execute immediately when the Active region is processed, before variable `b` is updated by a nonblocking assignment. If the same task is called from within a program, then the statement `S1` shall execute when the Reactive region is processed, after variable `b` might have been updated by nonblocking assignments. Statement `S2` always executes immediately after the delay expires; it does not wait for the Reactive region even though it was originally called from the program block.

16.5 Programwide space and anonymous programs

The set of program definitions and instances define a space of programwide data, tasks, and functions that is accessible only to programs.

Anonymous programs can be used inside packages or compilation-unit scopes (see 19.2 and 19.3) to declare items that are part of the programwide space without declaring a new scope. Items declared in an anonymous program share the same name space as the package or compilation-unit scope in which they are declared, but add the same semantic restrictions imposed on a program block.

NOTE—Although identifiers declared inside an anonymous program cannot be referenced outside any program block, attempting to declare another identifier with the same name outside the anonymous program block will generate an error. This occurs because the identifier shares the same name space within the scope of the surrounding package or compilation unit.

16.6 Program control tasks

In addition to the normal simulation control tasks ($stop and $finish), a program can use the $exit control task.

16.6.1 $exit()

Each program can be explicitly exited by calling the $exit system task. When all programs exit (implicitly or explicitly), the simulation finishes and an implicit call to $finish is made.

The syntax for the $exit system task is as follows:

```
task $exit();
```

When all **initial** blocks in a program finish (i.e., they execute their last statement), the program implicitly calls $exit. Calling $exit causes all processes spawned by the current program to be terminated.

17. Assertions

17.1 Introduction

NOTE—SystemVerilog adds features to specify assertions of a system. An assertion specifies a behavior of the system. Assertions are primarily used to validate the behavior of a design. In addition, assertions can be used to provide functional coverage and generate input stimulus for validation.

There are two kinds of assertions: concurrent and immediate.

— Immediate assertions follow simulation event semantics for their execution and are executed like a statement in a procedural block. Immediate assertions are primarily intended to be used with simulation.

— Concurrent assertions are based on clock semantics and use sampled values of variables. One of the goals of SystemVerilog assertions is to provide a common semantic meaning for assertions so that they can be used to drive various design and verification tools. Many tools, such as formal verification tools, evaluate circuit descriptions using cycle-based semantics, which typically relies on a clock signal or signals to drive the evaluation of the circuit. Any timing or event behavior between clock edges is abstracted away. Concurrent assertions incorporate this clock semantics. While this approach generally simplifies the evaluation of a circuit description, there are a number of scenarios under which this cycle-based evaluation provides different behavior from the standard event-based evaluation of SystemVerilog.

This clause describes both types of assertions.

17.2 Immediate assertions

The immediate assertion statement is a test of an expression performed when the statement is executed in the procedural code. The expression is nontemporal and is interpreted the same way as an expression in the condition of a procedural **if** statement. In other words, if the expression evaluates to X, Z, or 0, then it is interpreted as being false, and the assertion is said to fail. Otherwise, the expression is interpreted as being true, and the assertion is said to pass.

The immediate **assert** statement is a *statement_item* and can be specified anywhere a procedural statement is specified.

```
procedural_assertion_statement ::=                                          // from A.6.10
      ...
    | immediate_assert_statement
immediate_assert_statement ::=
    assert ( expression ) action_block
action_block ::=                                                            // from A.6.3
      statement _or_null
    | [ statement ] else statement
```

Syntax 17-1—Immediate assertion syntax (excerpt from Annex A)

The *action_block* specifies what actions are taken upon success or failure of the assertion. The statement associated with the success of the assert statement is the first statement. It is called the *pass statement* and is executed if the expression evaluates to true. The pass statement can, for example, record the number of successes for a coverage log, but can be omitted altogether. If the pass statement is omitted, then no user-specified action is taken when the assert expression is true. The statement associated with **else** is called a *fail statement* and is executed if the expression evaluates to false. The **else** statement can also be omitted. The action block is executed immediately after the evaluation of the assert expression.

The optional statement label (identifier and colon) creates a named block around the assertion statement (or any other SystemVerilog statement) and can be displayed using the `%m` format specification.

```
assert_foo : assert(foo) $display("%m passed"); else $display("%m failed");
```

NOTE—The assertion control system tasks are described in 22.8.

Because the assertion is a statement that something must be true, the failure of an assertion shall have a severity associated with it. By default, the severity of an assertion failure is *error*. Other severity levels can be specified by including one of the following severity system tasks in the fail statement:

— `$fatal` is a run-time fatal.
— `$error` is a run-time error.
— `$warning` is a run-time warning, which can be suppressed in a tool-specific manner.
— `$info` indicates that the assertion failure carries no specific severity.

The syntax for these system tasks is shown in 22.7.

If an assertion fails and no **else** clause is specified, the tool shall, by default, call `$error`, unless a tool-specific option, such as a command-line option, is enabled to suppress the failure.

All of these severity system tasks shall print a tool-specific message indicating the severity of the failure and specific information about the specific failure, which shall include the following information:

— The file name and line number of the assertion statement.
— The hierarchical name of the assertion, if it is labeled, or the scope of the assertion if it is not labeled.

For simulation tools, these tasks shall also include the simulation run time at which the severity system task is called.

Each system task can also include additional user-specified information using the same format as the Verilog `$display`.

If more than one of these system tasks is included in the **else** clause, then each shall be executed as specified.

If the severity system task is executed at a time other than when the assertion fails, the actual failure time of the assertion can be recorded and displayed programmatically. For example:

```
time t;

always @(posedge clk)
   if (state == REQ)
      assert (req1 || req2);
      else begin
         t = $time;
         #5 $error("assert failed at time %0t",t);
      end
```

If the assertion fails at time 10, the error message shall be printed at time 15, but the user-defined string printed shall be "assert failed at time 10".

The display of messages of warning and information types can be controlled by a tool-specific option, such as a command-line option.

Because the fail statement, like the pass statement, is any legal SystemVerilog procedural statement, it can also be used to signal a failure to another part of the testbench.

```
assert (myfunc(a,b)) count1 = count + 1; else ->event1;
assert (y == 0) else flag = 1;
```

17.3 Concurrent assertions overview

Concurrent assertions describe behavior that spans over time. Unlike immediate assertions, the evaluation model is based on a clock so that a concurrent assertion is evaluated only at the occurrence of a clock tick. The values of variables used in the evaluation are the sampled values. This way, a predictable result can be obtained from the evaluation, regardless of the simulator's internal mechanism of ordering events and evaluating events. This model of execution also corresponds to the synthesis model of hardware interpretation from an register transfer language (RTL) description.

The values of variables used in assertions are sampled in the Preponed region of a time slot, and the assertions are evaluated during the Observe region. If a variable used in an assertion is a **clocking** block input variable, the variable must be sampled by the **clocking** block with #1step sampling. Any other type of sampling for the **clocking** block variable shall result in an error. The assertion using the **clocking** block variable shall not do its own sampling on the variable, but rather use the sampled value produced by the **clocking** block. This is explained in Clause 9.

The timing model employed in a concurrent assertion specification is based on clock ticks and uses a generalized notion of clock cycles. The definition of a clock is explicitly specified by the user and can vary from one expression to another.

A clock tick is an atomic moment in time that itself spans no duration of time. A clock shall tick only once at any simulation time, and the sampled values for that simulation time are used for evaluation of concurrent assertions. In an assertion, the sampled value is the only valid value of a variable at a clock tick. Figure 17-1 shows the values of a variable as the clock progresses. The value of signal req is low at clock ticks 1 and 2. At clock tick 3, the value is sampled as high and remains high until clock tick 6. The sampled value of variable req at clock tick 6 is low and remains low up to and including clock tick 9. Notice that the simulation value transitions to high at clock tick 9. However, the sampled value at clock tick 9 is low.
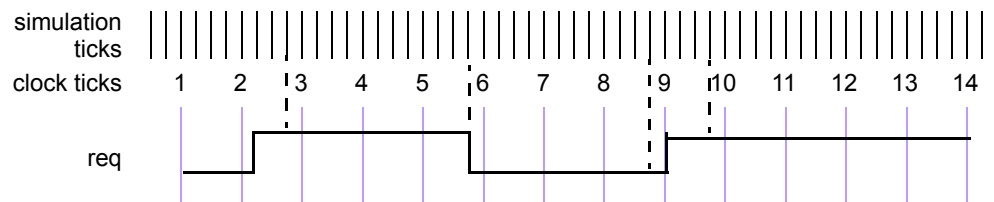
Figure 17-1—Sampling a variable on simulation ticks

An expression used in an assertion is always tied to a clock definition. The sampled values are used to evaluate value change expressions or boolean subexpressions that are required to determine a match of a sequence.

For concurrent assertions, the following statements apply:

— It is important to ensure that the defined clock behavior is glitch free. Otherwise, wrong values can be sampled.

— If a variable that appears in the expression for clock also appears in an expression with an assertion, the values of the two usages of the variable can be different. The current value of the variable is used in the clock expression, while the sampled value of the variable is used within the assertion.

The clock expression that controls evaluation of a sequence can be more complex than just a single signal name. Expressions such as (clk && gating_signal) and (clk **iff** gating_signal) can be used to represent a gated clock. Other more complex expressions are possible. However, in order to ensure proper behavior of the system and conform as closely as possible to truly cycle-based semantics, the signals in a clock expression must be glitch-free and should only transition once at any simulation time.

An example of a concurrent assertion is as follows:

```
base_rule1: assert property (cont_prop(rst,in1,in2)) $display("%m, passing");
            else $display("%m, failed");
```

The keyword **property** distinguishes a concurrent assertion from an immediate assertion. The syntax of concurrent assertions is discussed in 17.13.

17.4 Boolean expressions

The outcome of the evaluation of an expression is boolean and is interpreted the same way as an expression is interpreted in the condition of a procedural **if** statement. In other words, if the expression evaluates to X, Z, or 0, then it is interpreted as being false. Otherwise, it is true.

There are certain restrictions on the expressions that can appear in concurrent assertions. The restrictions on operand types, variables, and operators are specified in 17.4.1, 17.4.2, and 17.4.3.

Expressions are allowed to include function calls, but certain semantic restrictions are imposed.

— Functions that appear in expressions cannot contain output or **ref** arguments (**const ref** are allowed).

— Functions should be automatic (or preserve no state information) and have no side effects.

There are two places where Boolean expressions occur in concurrent properties:

— In the sequences used to build properties

— In the top-level **disable iff** clause (see 17.11)

The expressions used in defining sequences are evaluated over the sampled values of all variables (other than local variables as described in 17.8) and the current values of local variables and the sequence boolean methods ended and matched (see 17.12.6). The expression in the **disable iff** clause is evaluated using the current values of variables (not sampled) and can contain the sequence boolean method triggered. It must not contain any reference to local variables and the sequence methods ended and matched. If a sampled value function (see 17.7.3) is used in the expression, the sampling clock must be explicitly specified in the actual argument list. For example:

```
assert property ( @(posedge clk)
   disable iff (a && $rose(b, posedge clk)) trigger |=> test_expr );
```

The **disable iff** expression will preempt the evaluation of the assertion in a time step where a is 1 and the sampled value function returns a 1 as determined by the rules of evaluation for use outside sequences described in 17.7.3.

17.4.1 Operand types

The following types are not allowed:

— Noninteger types (**shortreal**, **real**, and **realtime**)

— **string**

- **event**
- **chandle**
- **class**
- Associative arrays
- Dynamic arrays

Fixed-size arrays, packed or unpacked, can be used as a whole or as part-selects or as indexed bit-selects or part-selects. The indices can be constants, parameters, or variables.

The following example shows some possible forms of comparison of members of structures and unions:

```
typedef int array [4];
typedef struct {int a, b, c, d} record;
union { record r; array a; } p, q;
```

The following comparisons are legal in expressions:

```
p.a == q.a
```

and

```
p.r == q.r
```

The following example provides further illustration of the use of arrays in expressions:

```
logic [7:0] arrayA [16], arrayB[16];
```

The following comparisons are legal:

```
arrayA == arrayB
arrayA != arrayB
arrayA[i] >= arrayB[j]
arrayB[i][j+:2] == arrayA[k][m-:2]
(arrayA[i] & (~arrayB[j])) == 0
```

17.4.2 Variables

The variables that can appear in expressions must be static design variables, function calls returning values of types described in 17.4.1, or local variables. Static variables declared in programs, interfaces, or **clocking** blocks can also be accessed. If a reference is to a static variable declared in a task, that variable is sampled as any other variable, independent of calls to the task.

17.4.3 Operators

All operators that are valid for the types described in 17.4.1 are allowed with the exception of assignment operators and increment and decrement operators. SystemVerilog includes the C assignment operators, such as +=, and the C increment and decrement operators, ++ and --. These operators cannot be used in expressions that appear in assertions. This restriction prevents side effects.

17.5 Sequences

```
sequence_expr ::=                                                          // from A.2.10
        cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      | expression_or_dist [ boolean_abbrev ]
      | ( expression_or_dist {, sequence_match_item } ) [ boolean_abbrev ]
      | sequence_instance [ sequence_abbrev ]
      | ( sequence_expr {, sequence_match_item } ) [ sequence_abbrev ]
      | sequence_expr and sequence_expr
      | sequence_expr intersect sequence_expr
      | sequence_expr or sequence_expr
      | first_match ( sequence_expr {, sequence_match_item} )
      | expression_or_dist throughout sequence_expr
      | sequence_expr within sequence_expr
      | clocking_event sequence_expr

cycle_delay_range ::=
        ## integral_number
      | ## identifier
      | ## ( constant_expression )
      | ## [ cycle_delay_const_range_expression ]

sequence_match_item ::=
        operator_assignment
      | inc_or_dec_expression
      | subroutine_call

sequence_instance ::=
        ps_sequence_identifier [ ( [ list_of_arguments ] ) ]

actual_arg_expr ::=
        event_expression
      | $

boolean_abbrev ::=
        consecutive_repetition
      | non_consecutive_repetition
      | goto_repetition

sequence_abbrev ::= consecutive_repetition

consecutive_repetition ::= [* const_or_range_expression ]

non_consecutive_repetition ::= [= const_or_range_expression ]

goto_repetition ::= [-> const_or_range_expression ]

const_or_range_expression ::=
        constant_expression
      | cycle_delay_const_range_expression

cycle_delay_const_range_expression ::=
        constant_expression : constant_expression
      | constant_expression : $

expression_or_dist ::= expression [ dist { dist_list } ]
```

Syntax 17-2—Sequence syntax (excerpt from Annex A)

Properties are often constructed out of sequential behaviors. The **sequence** feature provides the capability to build and manipulate sequential behaviors. The simplest sequential behaviors are linear. A linear sequence is a finite list of SystemVerilog boolean expressions in a linear order of increasing time. The linear sequence is said to match along a finite interval of consecutive clock ticks provided the first boolean expression evaluates to true at the first clock tick, the second boolean expression evaluates to true at the second clock tick, and so forth, up to and including the last boolean expression evaluating to true at the last clock tick. A single boolean expression is an example of a simple linear sequence, and it matches at a single clock tick provided the boolean expression evaluates to true at that clock tick.

.

More complex sequential behaviors are described by SystemVerilog sequences. A sequence is a regular expression over the SystemVerilog boolean expressions that concisely specifies a set of zero, finitely many, or infinitely many linear sequences. If at least one of the linear sequences from this set matches along a finite interval of consecutive clock ticks, then the sequence is said to match along that interval.

A property may involve checking of one or more sequential behaviors beginning at various times. An attempted evaluation of a sequence is a search for a match of the sequence beginning at a particular clock tick. To determine whether such a match exists, appropriate boolean expressions are evaluated beginning at the particular clock tick and continuing at each successive clock tick until either a match is found or it is deduced that no match can exist.

Sequences can be composed by concatenation, analogous to a concatenation of lists. The concatenation specifies a delay, using ##, from the end of the first sequence until the beginning of the second sequence.

The following is the syntax for sequence concatenation.

```
sequence_expr ::=                                                        // from A.2.10
        cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
      ...
cycle_delay_range ::=
        ## integral_number
      | ## identifier
      | ## ( constant_expression )
      | ## [ cycle_delay_const_range_expression ]
cycle_delay_const_range_expression ::=
        constant_expression : constant_expression
      | constant_expression : $
```

Syntax 17-3—Sequence concatenation syntax (excerpt from Annex A)

In this syntax, the following statements apply:

— *constant_expression* is computed at compile time and must result in an integer value.

— *constant_expression* can only be 0 or greater.

— The **$** token is used to indicate the end of simulation. For formal verification tools, **$** is used to indicate a finite, but unbounded, range.

— When a range is specified with two expressions, the second expression must be greater than or equal to the first expression.

The context in which a sequence occurs determines when the sequence is evaluated. The first expression in a sequence is checked at the first occurrence of the clock tick at or after the expression that triggered evaluation of the sequence. Each successive element (if any) in the sequence is checked at the next subsequent occurrence of the clock.

A ## followed by a number or range specifies the delay from the current clock tick to the beginning of the sequence that follows. The delay ##1 indicates that the beginning of the sequence that follows is one clock tick later than the current clock tick. The delay ##0 indicates that the beginning of the sequence that follows is at the same clock tick as the current clock tick.

When used as a concatenation between two sequences, the delay is from the end of the first sequence to the beginning of the second sequence. The delay ##1 indicates that the beginning of the second sequence is one clock tick later than the end of the first sequence. The delay ##0 indicates that the beginning of the second sequence is at the same clock tick as the end of the first sequence.

The following are examples of delay expressions. `true is a boolean expression that always evaluates to true and is used for visual clarity. It can be defined as follows:

```
`define true 1

##0 a        // means a
##1 a        // means `true ##1 a
##2 a        // means `true ##1 `true ##1 a
##[0:3]a     // means (a) or (`true ##1 a) or (`true ##1 `true ##1 a) or
             //       (`true ##1 `true ##1 `true ##1 a)
a ##2 b // means a ##1 `true ##1 b
```

The sequence

```
req ##1 gnt ##1 !req
```

specifies that req be true on the current clock tick, gnt shall be true on the first subsequent tick, and req shall be false on the next clock tick after that. The ##1 operator specifies one clock tick separation. A delay of more than one clock tick can be specified, as in the following:

```
req ##2 gnt
```

This specifies that req shall be true on the current clock tick, and gnt shall be true on the second subsequent clock tick, as shown in Figure 17-2.
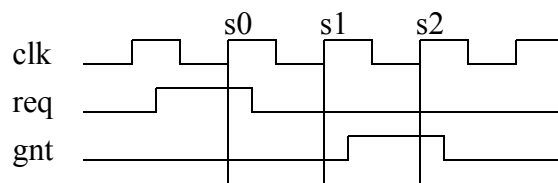
Figure 17-2—Concatenation of sequences

The following specifies that signal b shall be true on the Nth clock tick after signal a:

```
a ##N b     // check b on the Nth sample
```

To specify a concatenation of overlapped sequences, where the end point of one sequence coincides with the start of the next sequence, a value of 0 is used, as shown below.

```
a ##1 b ##1 c // first sequence seq1
d ##1 e ##1 f // second sequence seq2
(a ##1 b ##1 c) ##0 (d ##1 e ##1 f) // overlapped concatenation
```

In the above example, `c` must be true at the end point of sequence `seq1`, and `d` must be true at the start of sequence `seq2`. When concatenated with `0` clock tick delay, `c` and `d` must be true at the same time, resulting in a concatenated sequence equivalent to the following:

```
a ##1 b ##1 c&&d ##1 e ##1 f
```

It should be noted that no other form of overlapping between the sequences can be expressed using the concatenation operation.

In cases where the delay can be any value in a range, a time window can be specified as follows:

```
req ##[4:32] gnt
```

In the above case, signal `req` must be true at the current clock tick, and signal `gnt` must be true at some clock tick between the 4th and the 32nd clock tick after the current clock tick.

The time window can extend to a finite, but unbounded, range by using `$` as in the example below.

```
req ##[4:$] gnt
```

A sequence can be unconditionally extended by concatenation with `'true`.

```
a ##1 b ##1 c ##3 'true
```

After satisfying signal `c`, the sequence length is extended by three clock ticks. Such adjustments in the length of sequences can be required when complex sequences are constructed by combining simpler sequences.

17.6 Declaring sequences

A **sequence** can be declared in
— A module
— An interface
— A program
— A **clocking** block
— A package
— A compilation-unit scope

Sequences are declared using the following syntax:

```
concurrent_assertion_item_declaration ::=                                    // from A.2.10
    ...
    | sequence_declaration
sequence_declaration ::=
    sequence sequence_identifier [ ( [ tf_port_list ] ) ] ;
        { assertion_variable_declaration }
        sequence_expr ;
    endsequence [ : sequence_identifier ]
sequence_instance ::=
    ps_sequence_identifier [ ( [ list_of_arguments ] ) ]
actual_arg_expr ::=
    event_expression
    | $
assertion_variable_declaration ::=
    var_data_type list_of_variable_identifiers ;
```

Syntax 17-4—Declaring sequence syntax (excerpt from Annex A)

The `clocking_event` specifies the clock for the sequence.

A sequence is declared with optional formal arguments. When a sequence is instantiated, actual arguments can be passed to the sequence. The sequence gets expanded with the actual arguments by replacing the formal arguments with the actual arguments. Semantic checks are performed to ensure that the expanded sequence with the actual arguments is legal.

An actual argument can replace any of the following:
— Identifier
— Expression
— Event control expression
— Upper delay range or repetition range if the actual argument is $

Variables used in a sequence that are not formal arguments to the sequence are resolved according to the scoping rules from the scope in which the sequence is declared.

```
sequence s1;
    @(posedge clk) a ##1 b ##1 c;
endsequence
sequence s2;
    @(posedge clk) d ##1 e ##1 f;
endsequence
sequence s3;
    @(negedge clk) g ##1 h ##1 i;
endsequence
```

In this example, sequences `s1` and `s2` are evaluated on successive **posedge** events of `clk`. The sequence `s3` is evaluated on successive **negedge** events of `clk`.

Another example of sequence declaration, which includes arguments, is shown below:

```
sequence s20_1(data,en);
    (!frame && (data==data_bus)) ##1 (c_be[0:3] == en);
endsequence
```

Sequence `s20_1` does not specify a clock. In this case, a clock would be inherited from some external source, such as a **property** or an **assert** statement. A sequence can be referred to by its name. A hierarchical name can be used, consistent with the SystemVerilog naming conventions. A sequence can be referenced in a **sequence**, a **property**, a concurrent **assert**, an **assume** or a **cover** statement.

To use a named sequence as a subsequence of another sequence, simply reference its name. The evaluation of a sequence that references a named sequence is performed in the same way as if the named sequence was contained as a lexical part of the referencing sequence, with the formal arguments of the named sequence replaced by the actual ones and the remaining variables in the named sequence resolved according to the scope of the declaration of the named sequence. An example is shown below:

```
sequence s;
    a ##1 b ##1 c;
endsequence
sequence rule;
    @(posedge sysclk)
    trans ##1 start_trans ##1 s ##1 end_trans;
endsequence
```

Sequence `rule` in the preceding example is equivalent to the following:

```
sequence rule;
    @(posedge sysclk)
    trans ##1 start_trans ##1 a ##1 b ##1 c ##1 end_trans ;
endsequence
```

Any form of syntactic cyclic dependency of the sequence names is disallowed. The example below illustrates an illegal dependency of `s1` on `s2` and `s2` on `s1` because it creates a cyclic dependency.

```
sequence s1;
    @(posedge sysclk) (x ##1 s2);
endsequence
sequence s2;
    @(posedge sysclk) (y ##1 s1);
endsequence
```

17.6.1 Typed formal arguments in sequence declarations

Formal arguments of sequences can optionally be typed. To declare a type for a formal argument of a sequence, it is required to prefix the argument with a type. A formal argument that is not prefixed by a type will be untyped.

Exporting values of local variables through typed formal arguments is not supported.

The supported data types for sequence formal arguments are the types that are allowed for operands in assertion expressions (see 17.4.1). The assignment rules for assigning actual argument expressions to formal arguments, at the time of sequence instantiation, are the same as the general rules for doing assignment of a typed variable with a typed expression (see Clause 4).

For example, two equivalent ways of passing arguments are shown below. The first has untyped arguments, and the second has typed arguments:

```
sequence rule6_with_no_type(x, y);
    ##1 x ##[2:10] y;
endsequence
```

```
sequence rule6_with_type(bit x, bit y);
   ##1 x ##[2:10] y;
endsequence
```

Another example, in which a local variable is used to sample a formal argument, shows how to get the effect of "pass by value". Pass by value is not currently supported as a mode of argument passing.

```
sequence foo(bit a, bit b);
   bit loc_a;
   (1'b1, loc_a = a) ##0
   (t == loc_a) [*0:$] ##1 b;
endsequence
```

17.7 Sequence operations

17.7.1 Operator precedence

Operator precedence and associativity are listed in Table 17-1, below. The highest precedence is listed first.

Table 17-1—Operator precedence and associativity

SystemVerilog expression operators	Associativity
`[* ]  [= ]  [-> ]`	—
`##`	Left
`throughout`	Right
`within`	Left
`intersect`	Left
`and`	Left
`or`	Left

17.7.2 Repetition in sequences

Following is the syntax for sequence repetition.

```
sequence_expr ::=                                                    // from A.2.10
      ...
      | expression_or_dist [ boolean_abbrev ]
      | ( expression_or_dist {, sequence_match_item } ) [ boolean_abbrev ]
      | sequence_instance [ sequence_abbrev ]
      | ( sequence_expr {, sequence_match_item} ) [ sequence_abbrev ]
      ...
boolean_abbrev ::=
      consecutive_repetition
      | non_consecutive_repetition
      | goto_repetition
sequence_abbrev ::= consecutive_repetition
consecutive_repetition ::= [* const_or_range_expression ]
non_consecutive_repetition ::= [= const_or_range_expression ]
goto_repetition ::= [-> const_or_range_expression ]
const_or_range_expression ::=
      constant_expression
      | cycle_delay_const_range_expression
cycle_delay_const_range_expression ::=
      constant_expression : constant_expression
      | constant_expression : $
```

Syntax 17-5—Sequence repetition syntax (excerpt from Annex A)

The number of iterations of a repetition can either be specified by exact count or be required to fall within a finite range. If specified by exact count, then the number of iterations is defined by a non-negative integer constant expression. If required to fall within a finite range, then the minimum number of iterations is defined by a non-negative integer constant expression; and the maximum number of iterations either is defined by a non-negative integer constant expression or is $, indicating a finite, but unbounded, maximum.

If both the minimum and maximum numbers of iterations are defined by non-negative integer constant expressions, then the minimum number must be less than or equal to the maximum number.

Three kinds of repetition are provided:

— *Consecutive repetition* ([*): Consecutive repetition specifies finitely many iterative matches of the operand sequence, with a delay of one clock tick from the end of one match to the beginning of the next. The overall repetition sequence matches at the end of the last iterative match of the operand.

— *Goto repetition* ([->): Goto repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at the last iterative match of the operand.

— *Nonconsecutive repetition* ([=): Nonconsecutive repetition specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and no match of the operand strictly in between. The overall repetition sequence matches at or after the last iterative match of the operand, but before any later match of the operand.

The effect of consecutive repetition of a subsequence within a sequence can be achieved by explicitly iterating the subsequence, as follows:

```
a ##1 b ##1 b ##1 b ##1 c
```

Using the consecutive repetition operator `[*3]`, which indicates three iterations, this sequential behavior is specified more succinctly:

```
a ##1 b [*3] ##1 c
```

A consecutive repetition specifies that the operand sequence must match a specified number of times. The consecutive repetition operator `[*N]` specifies that the operand sequence must match *N* times in succession. For example:

```
a [*3] means a ##1 a ##1 a
```

Using `0` as the repetition number, an empty sequence results, as follows:

```
a [*0]
```

An empty sequence is one that does not match over any positive number of clock ticks. The following rules apply for concatenating sequences with empty sequences. An empty sequence is denoted as *empty*, and a sequence is denoted as *seq*.

— `(empty ##0 seq)` does not result in a match.
— `(seq ##0 empty)` does not result in a match.
— `(empty ##n seq)`, where n is greater than 0, is equivalent to (##(n-1) seq).
— `(seq ##n empty)`, where n is greater than 0, is equivalent to (seq ##(n-1) 'true).

For example:

```
b ##1 ( a[*0] ##0 c)
```

produces no match of the sequence.

```
b ##1 a[*0:1] ##2 c
```

is equivalent to

```
(b ##2 c) or (b ##1 a ##2 c)
```

The syntax allows combination of a delay and repetition in the same sequence. The following are both allowed:

```
`true ##3 (a [*3])     // means `true ##1 `true ##1 `true ##1 a ##1 a ##1 a
(`true ##2 a) [*3]     // means (`true ##2 a) ##1 (`true ##2 a) ##1
                       // (`true ##2 a), which in turn means `true ##1 `true ##1
                       // a ##1 `true ##1 `true ##1 a ##1 `true ##1 `true ##1 a
```

A sequence can be repeated as follows:

```
(a ##2 b) [*5]
```

This is the same as the following:

```
(a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b)
```

A repetition with a range of minimum `min` and maximum `max` number of iterations can be expressed with the consecutive repetition operator `[* min:max]`.

For example:

```
(a ##2 b)[*1:5]
```

is equivalent to

```
    (a ##2 b)
or (a ##2 b ##1 a ##2 b)
or (a ##2 b ##1 a ##2 b ##1 a ##2 b)
or (a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b)
or (a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b ##1 a ##2 b)
```

Similarly,

```
(a[*0:3] ##1 b ##1 c)
```

is equivalent to

```
    (b ##1 c)
or (a ##1 b ##1 c)
or (a ##1 a ##1 b ##1 c)
or (a ##1 a ##1 a ##1 b ##1 c)
```

To specify a finite, but unbounded, number of iterations, the dollar sign ($) is used. For example, the repetition

```
a ##1 b [*1:$] ##1 c
```

matches over an interval of three or more consecutive clock ticks if a is true on the first clock tick, c is true on the last clock tick, and b is true at every clock tick strictly in between the first and the last.

Specifying the number of iterations of a repetition by exact count is equivalent to specifying a range in which the minimum number of repetitions is equal to the maximum number of repetitions. In other words, seq[*n] is equivalent to seq[*n:n].

The goto repetition (nonconsecutive exact repetition) takes a boolean expression rather than a sequence as operand. It specifies the iterative matching of the boolean expression at clock ticks that are not necessarily consecutive and ends at the last iterative match. For example:

```
a ##1 b [->2:10] ##1 c
```

matches over an interval of consecutive clock ticks provided a is true on the first clock tick, c is true on the last clock tick, b is true on the penultimate clock tick, and, including the penultimate, there are at least 2 and at most 10 not necessarily consecutive clock ticks strictly in between the first and last on which b is true. This sequence is equivalent to

```
a ##1 ((!b[*0:$] ##1 b) [*2:10]) ##1 c
```

The nonconsecutive repetition is like the goto repetition except that a match does not have to end at the last iterative match of the operand boolean expression. The use of nonconsecutive repetition instead of goto repetition allows the match to be extended by arbitrarily many clock ticks provided the boolean expression is false on all of the extra clock ticks. For example:

```
a ##1 b [=2:10] ##1 c
```

matches over an interval of consecutive clock ticks provided `a` is true on the first clock tick, `c` is true on the last clock tick, and there are at least 2 and at most 10 not necessarily consecutive clock ticks strictly in between the first and last on which `b` is true. This sequence is equivalent to the following:

```
a ##1 ((!b [*0:$] ##1 b) [*2:10]) ##1 !b[*0:$] ##1 c
```

17.7.3 Sampled value functions

This subclause describes the system functions available for accessing sampled values of an expression. These functions include the capability to access current sampled value, access sampled value in the past, or detect changes in sampled value of an expression. Sampling of an expression is explained in 17.3. The sequence methods `ended`, `triggered`, and `matched` cannot be used as arguments to these functions. The following functions are provided:

```
$sampled(expression [, clocking_event])

$rose( expression [, clocking_event])

$fell( expression [, clocking_event])

$stable( expression [, clocking_event])

$past( expression1 [, number_of_ticks] [, expression2] [, clocking_event])
```

The use of these functions is not limited to assertion features; they can be used as expressions in procedural code as well. The clocking event, although optional as an explicit argument to the functions, is required for their semantics. The clocking event is used to sample the value of the argument expression.

The clocking event must be explicitly specified as an argument or inferred from the code where it is used. The following rules are used to infer the clocking event:

— If used in an assertion, the appropriate clocking event from the assertion is used.
— If used in an action block of a singly clocked assertion, the clock of the assertion is used.
— If used in a procedural block, the inferred clock, if any, for the procedural code (see 17.13.5) is used.

Otherwise, default clocking (see 15.11) is used.

When these functions are used in an assertion, the clocking event argument of the functions, if specified, shall be identical to the clocking event of the expression in the assertion. In the case of multiclock assertions, the appropriate clocking event for the expression where the function is used is applied to the function.

Function `$sampled` returns the sampled value of the expression with respect to the last occurrence of the clocking event. When `$sampled` is invoked prior to the occurrence of the first clocking event, the value of X is returned. The use of `$sampled` in assertions, although allowed, is redundant, as the result of the function is identical to the sampled value of the expression itself used in the assertion.

Three functions are provided to detect changes in sampled values: `$rose`, `$fell`, and `$stable`.

A value change function detects the change in the sampled value of an expression. The clocking event is used to obtain the sampled value of the argument expression at a clock tick prior to the current simulation time unit. Here, the current simulation time unit refers to the simulation time unit in which the function is evaluated. This sampled value is compared against the value of the expression determined at the prepone time of the current simulation time unit. The result of a value change expression is true or false and can be used as a boolean expression.

— `$rose` returns true if the LSB of the expression changed to `1`. Otherwise, it returns false.

— $fell returns true if the LSB of the expression changed to 0. Otherwise, it returns false.

— $stable returns true if the value of the expression did not change. Otherwise, it returns false.

When these functions are called at or before the first clock tick of the clocking event, the results are computed by comparing the current sampled value of the expression to X.

Figure 17-3 illustrates two examples of value changes:

— Value change expression e1 is defined as $rose(req).

— Value change expression e2 is defined as $fell(ack).
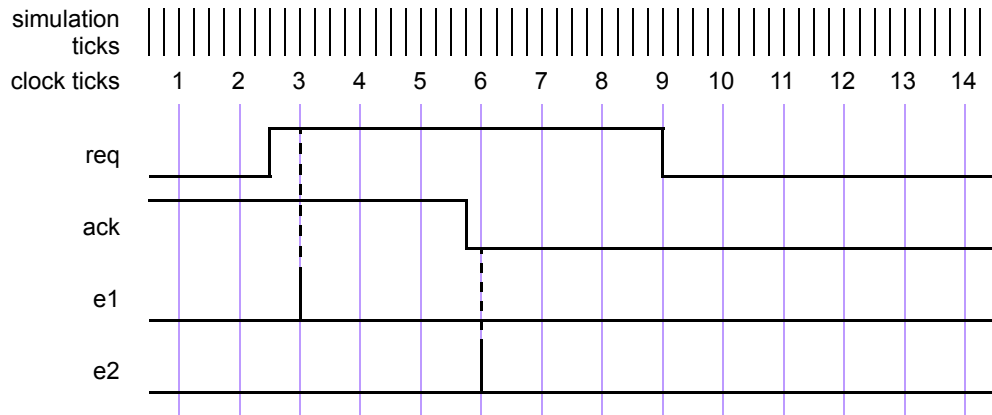
Figure 17-3—Value change expressions

The clock ticks used for sampling the variables are derived from the clock for the property, which is different from the simulation ticks. Assume, for now, that this clock is defined elsewhere. At clock tick 3, e1 occurs because the value of req at clock tick 2 was low and the value at clock tick 3 is high. Similarly, e2 occurs at clock tick 6 because the value of ack was sampled as high at clock tick 5 and sampled as low at clock tick 6.

The example below illustrates the use of $rose in SystemVerilog code outside assertions.

```
always @(posedge clk)
    reg1 <= a & $rose(b);
```

In this example, the clocking event (posedge clk) is applied to $rose. $rose is true whenever the sampled value of b changed to 1 from its sampled value at the previous tick of the clocking event.

In addition to accessing value changes, the past values can be accessed with the $past function. The following three optional arguments are provided:

— expression2 is used as a gating expression for the clocking event.

— number_of_ticks specifies the number of clock ticks in the past.

— clocking_event specifies the clocking event for sampling expression1.

expression1 and expression2 can be any expression allowed in assertions.

number_of_ticks must be 1 or greater. If number_of_ticks is not specified, then it defaults to 1. $past returns the sampled value of the expression that was present number_of_ticks prior to the time of

evaluation of $past. A clock tick is based on `clocking_event`. If the specified clock tick in the past is before the start of simulation, the returned value from the $past function is a value of X.

The optional argument `clocking_event` specifies the clock for the function. The rules governing the usage of `clocking_event` are same as those described for the value change function.

When intermediate optional arguments between two arguments are not needed, a comma must be placed for each omitted argument. For example:

```
$past(in1, , enable);
```

Here, a comma is specified to omit `number_of_ticks`. The default of 1 is used for the empty `number_of_ticks` argument. There is no need to include a comma for the omitted `clocking_event` argument, as it does not fall within the specified arguments.

$past can be used in any SystemVerilog expression. An example is shown below.

```
always @(posedge clk)
  reg1 <= a & $past(b);
```

In this example, the clocking event (**posedge** clk) is applied to $past. $past is evaluated in the current occurrence of (**posedge** clk) and returns the value of b sampled at the previous occurrence of (**posedge** clk).

When `expression2` is specified, the sampling of `expression1` is performed based on its clock gated with `expression2`. For example:

```
always @(posedge clk)
  if (enable) q <= d;

always @(posedge clk)
assert property (done |=> (out == $past(q, 2,enable)) );
```

In this example, the sampling of q for evaluating $past is based on the clocking expression

```
posedge clk iff enable
```

17.7.4 AND operation

The binary operator **and** is used when both operands are expected to match, but the end times of the operand sequences can be different.

sequence_expr ::=	// from *A.2.10*
...	
| sequence_expr **and** sequence_expr	

Syntax 17-6—And operator syntax (excerpt from Annex A)

The two operands of **and** are sequences. The requirement for the match of the **and** operation is that both the operands must match. The operand sequences start at the same time. When one of the operand sequences matches, it waits for the other to match. The end time of the composite sequence is the end time of the operand sequence that completes last.

When `te1` and `te2` are sequences, then the composite sequence

```
    te1 and te2
```

matches if `te1` and `te2` match. The end time is the end time of either `te1` or `te2`, whichever matches last.

The following example is a sequence with operator **and**, where the two operands are sequences:

```
    (te1 ##2 te2) and (te3 ##2 te4 ##2 te5)
```

The operation as illustrated in Figure 17-4 shows the evaluation attempt at clock tick 8. Here, the two operand sequences are (te1 ##2 te2) and (te3 ##2 te4 ##2 te5). The first operand sequence requires that first `te1` evaluates to true followed by `te2` two clock ticks later. The second sequence requires that first `te3` evaluates to true followed by `te4` two clock ticks later, followed by `te5` two clock ticks later.
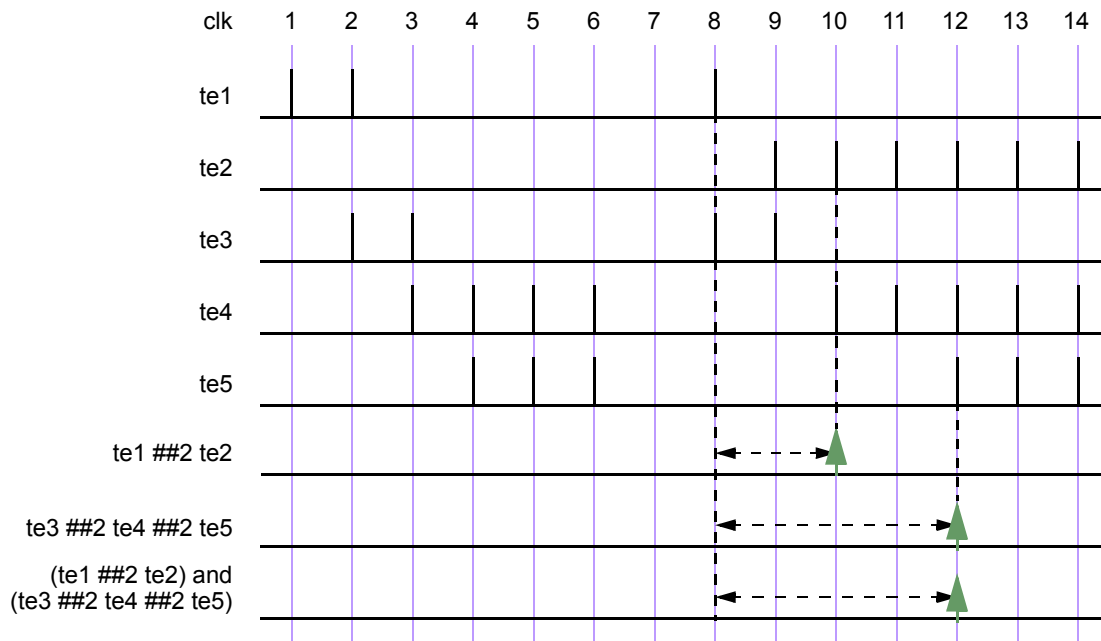
Figure 17-4—ANDing (and) two sequences

This attempt results in a match because both operand sequences match. The end times of matches for the individual sequences are clock ticks 10 and 12. The end time for the composite sequence is the later of the two end times; therefore, a match is recognized for the composite sequence at clock tick 12.

In the following example, the first operand sequence has a concatenation operator with range from 1 to 5:

```
    (te1 ##[1:5] te2) and (te3 ##2 te4 ##2 te5)
```

The first operand sequence requires that `te1` evaluate to true and that `te2` evaluate to true 1, 2, 3, 4, or 5 clock ticks later. The second operand sequence is the same as in the previous example. To consider all possibilities of a match of the composite sequence, the following steps can be taken:

a)   Five threads of evaluation are started for the five possible linear sequences associated with the first sequence operand.

b)   The second operand sequence has only one associated linear sequence; therefore, only one thread of evaluation is started for it.

c) Figure 17-5 shows the evaluation attempt beginning at clock tick 8. All five linear sequences for the first operand sequence match, as shown in a time window; therefore, there are five matches of the first operand sequence, ending at clock ticks 9, 10, 11, 12, and 13, respectively. The second operand sequence matches at clock tick 12.

d) Each match of the first operand sequence is combined with the single match of the second operand sequence, and the rules of the AND operation determine the end time of the resulting match of the composite sequence.

The result of this computation is five matches of the composite sequence, four of them ending at clock tick 12, and the fifth ending at clock tick 13. Figure 17-5 shows the matches of the composite sequence ending at clock ticks 12 and 13.
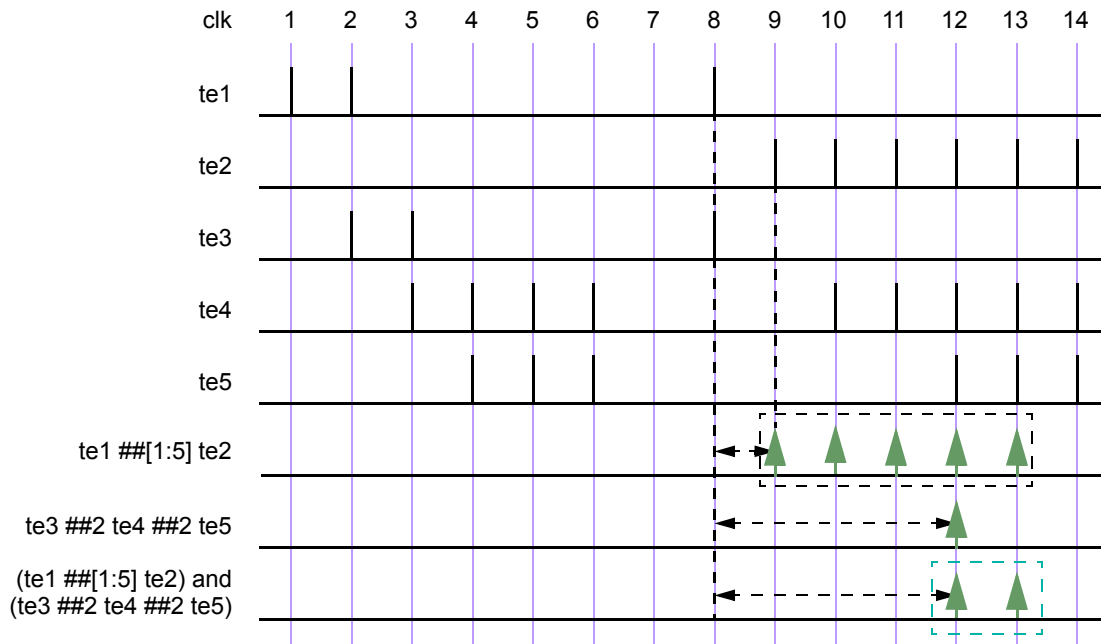
Figure 17-5—ANDing (and) two sequences, including a time range

If te1 and te2 are sampled expressions (not sequences), the sequence (te1 **and** te2) matches if te1 and te2 both evaluate to true.

An example is illustrated in Figure 17-6, which shows the results for attempts at every clock tick. The sequence matches at clock tick 1, 3, 8, and 14 because both te1 and te2 are simultaneously true. At all other clock ticks, match of the AND operation fails because either te1 or te2 is false.
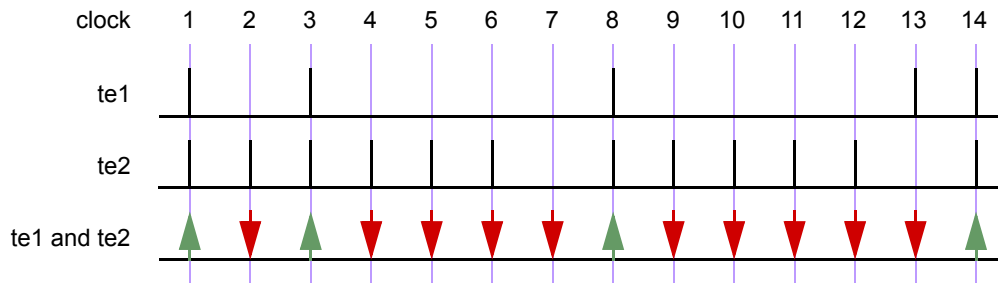
Figure 17-6—ANDing (and) two boolean expressions

17.7.5 Intersection (AND with length restriction)

The binary operator **intersect** is used when both operand sequences are expected to match, and the end times of the operand sequences must be the same.

```
sequence_expr ::=                                                        // from A.2.10
    ...
    | sequence_expr intersect sequence_expr
```

Syntax 17-7—Intersect operator syntax (excerpt from Annex A)

The two operands of **intersect** are sequences. The requirements for match of the **intersect** operation are as follows:

— Both the operands must match.

— The lengths of the two matches of the operand sequences must be the same.

The additional requirement on the length of the sequences is the basic difference between **and** and **intersect**.

An attempted evaluation of an **intersect** sequence can result in multiple matches. The results of such an attempt can be computed as follows:

— Matches of the first and second operands that are of the same length are paired. Each such pair results in a match of the composite sequence, with length and end point equal to the shared length and end point of the paired matches of the operand sequences.

— If no such pair is found, then there is no match of the composite sequence.

Figure 17-7 is similar to Figure 17-5, except that **and** is replaced by **intersect**. In this case, unlike in Figure 17-5, there is only a single match at clock tick 12.
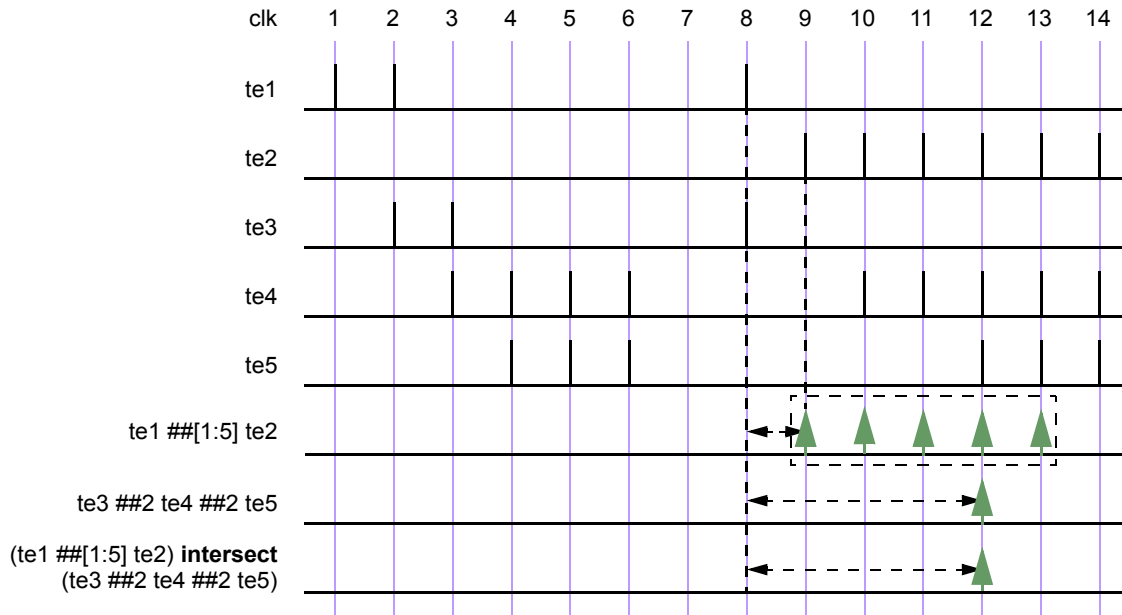
Figure 17-7—Intersecting two sequences

17.7.6 OR operation

The operator **or** is used when at least one of the two operand sequences is expected to match.

sequence_expr ::=   *// from A.2.10*
    ...
    | sequence_expr **or** sequence_expr

Syntax 17-8—Or operator syntax (excerpt from Annex A)

The two operands of **or** are sequences.

If the operands te1 and te2 are expressions, then

    te1 **or** te2

matches at any clock tick on which at least one of te1 and te2 evaluates to true.

Figure 17-8 illustrates an OR operation for which the operands te1 and te2 are expressions. The composite sequence does not match at clock ticks 7 and 13 because te1 and te2 are both false at those times. At all other clock ticks, the composite sequence matches, as at least one of the two operands evaluates to true.
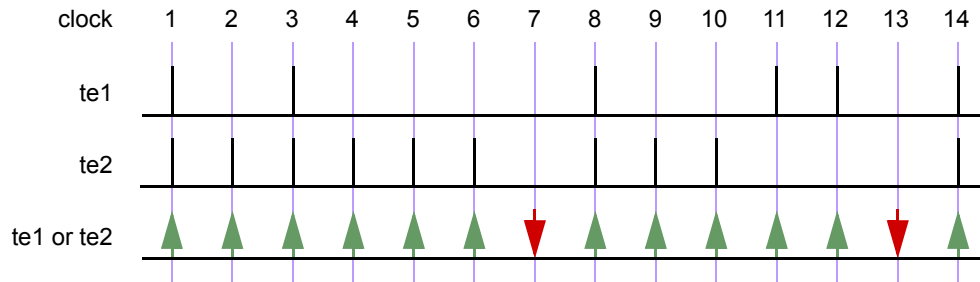
Figure 17-8—ORing (or) two sequences

When `te1` and `te2` are sequences, then the sequence

    te1 **or** te2

matches if at least one of the two operand sequences `te1` and `te2` matches. Each match of either `te1` or `te2` constitutes a match of the composite sequence, and its end time as a match of the composite sequence is the same as its end time as a match of `te1` or of `te2`. In other words, the set of matches of `te1` or `te2` is the union of the set of matches of `te1` with the set of matches of `te2`.

The following example shows a sequence with operator **or** where the two operands are sequences. Figure 17-9 illustrates this example.

    (te1 ##2 te2) **or** (te3 ##2 te4 ##2 te5)
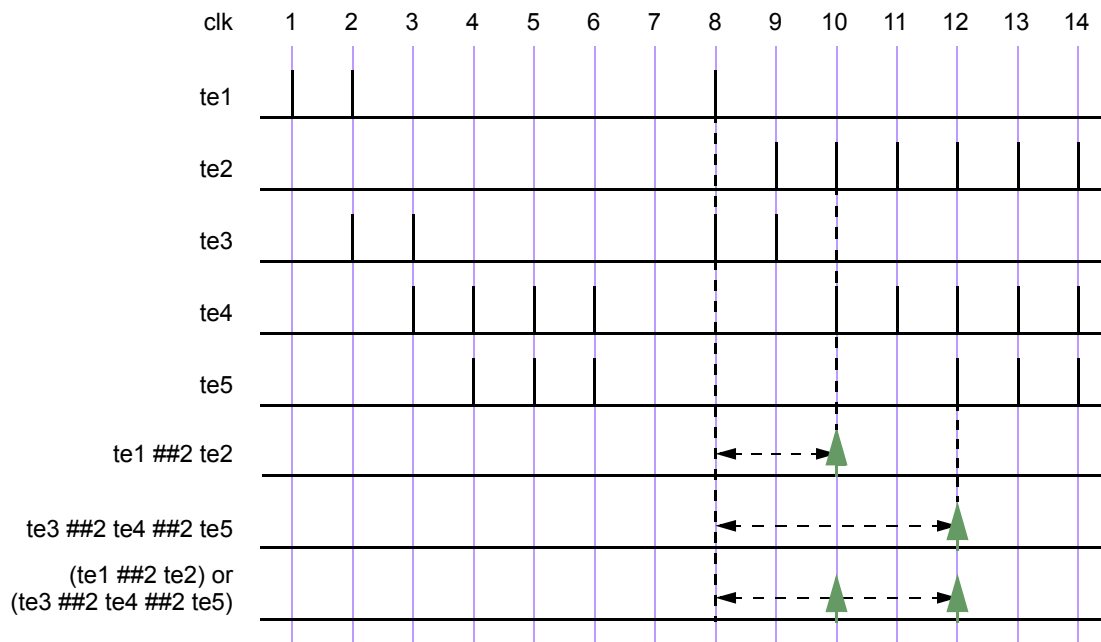
Figure 17-9—ORing (or) two sequences

Here, the two operand sequences are `(te1 ##2 te2)` and `(te3 ##2 te4 ##2 te5)`. The first sequence requires that `te1` first evaluates to true, followed by `te2` two clock ticks later. The second sequence requires that `te3` evaluates to true, followed by `te4` two clock ticks later, followed by `te5` two clock ticks later. In

Figure 17-9, the evaluation attempt for clock tick 8 is shown. The first sequence matches at clock tick 10, and the second sequence matches at clock tick 12. Therefore, two matches for the composite sequence are recognized.

In the following example, the first operand sequence has a concatenation operator with range from 1 to 5:

```
(te1 ##[1:5] te2) or (te3 ##2 te4 ##2 te5)
```

The first operand sequence requires that `te1` evaluate to true and that `te2` evaluate to true 1, 2, 3, 4, or 5 clock ticks later. The second operand sequence requires that `te3` evaluate to true, that `te4` evaluate to true two clock ticks later, and that `te5` evaluate to true another two clock ticks later. The composite sequence matches at any clock tick on which at least one of the operand sequences matches. As shown in Figure 17-10, for the attempt at clock tick 8, the first operand sequence matches at clock ticks 9, 10, 11, 12, and 13, while the second operand matches at clock tick 12. The composite sequence, therefore, has one match at each of clock ticks 9, 10, 11, and 13 and has two matches at clock tick 12.
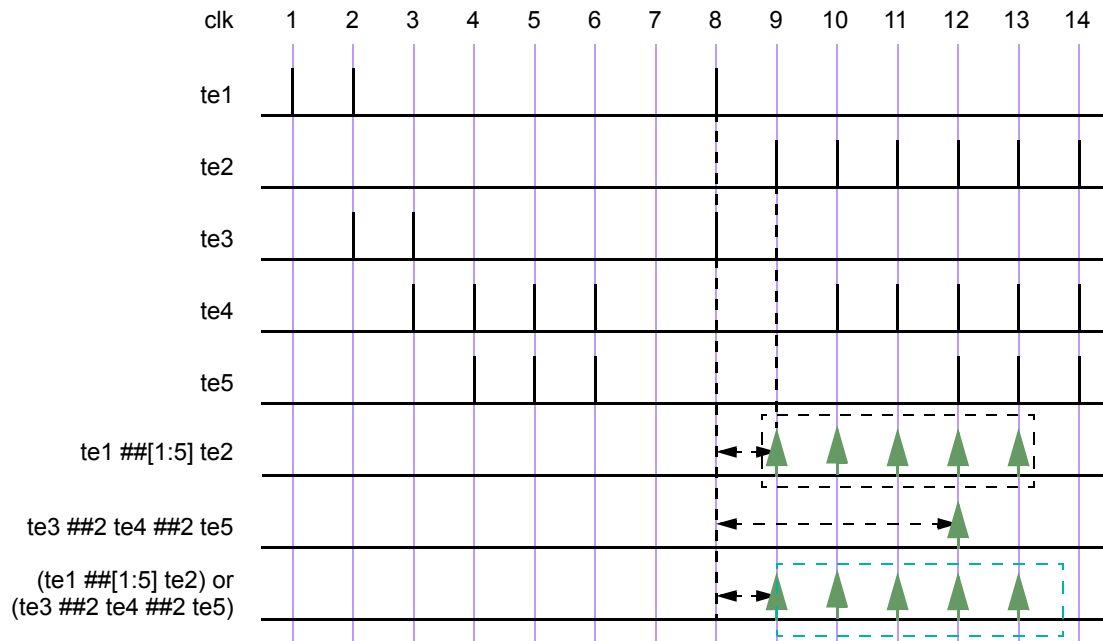
Figure 17-10—ORing (or) two sequences, including a time range

17.7.7 First_match operation

The **first_match** operator matches only the first of possibly multiple matches for an evaluation attempt of its operand sequence. This allows all subsequent matches to be discarded from consideration. In particular, when a sequence is a subsequence of a larger sequence, then applying the **first_match** operator has significant effect on the evaluation of the enclosing sequence.

```
sequence_expr ::=                                                    // from A.2.10
    ...
    | first_match ( sequence_expr {, sequence_match_item} )
```

Syntax 17-9—First_match operator syntax (excerpt from Annex A)

An evaluation attempt of **first_match** (*seq*) results in an evaluation attempt for the operand *seq* beginning at the same clock tick. If the evaluation attempt for *seq* produces no match, then the evaluation attempt for **first_match** (*seq*) produces no match. Otherwise, the match of *seq* with earliest ending clock tick is a match of **first_match** (*seq*). If there are multiple matches of *seq* with the same ending clock tick as the earliest one, then all those matches are matches of **first_match** (*seq*).

The example below shows a variable delay specification.

```
sequence t1;
    te1 ## [2:5] te2;
endsequence
sequence ts1;
    first_match(te1 ## [2:5] te2);
endsequence
```

Here, te1 and te2 are expressions. Each attempt of sequence t1 can result in matches for up to four of the following sequences:

```
te1 ##2 te2
te1 ##3 te2
te1 ##4 te2
te1 ##5 te2
```

However, sequence ts1 can result in a match for only one of the above four sequences. Whichever match of the above four sequences ends first is a match of sequence ts1.

For example:

```
sequence t2;
    (a ##[2:3] b) or (c ##[1:2] d);
endsequence
sequence ts2;
    first_match(t2);
endsequence
```

Each attempt of sequence t2 can result in matches for up to four of the following sequences:

```
a ##2 b
a ##3 b
c ##1 d
c ##2 d
```

Sequence ts2 matches only the earliest ending match of these sequences. If a, b, c, and d are expressions, then it is possible to have matches ending at the same time for both.

```
a ##2 b
c ##2 d
```

If both of these sequences match and (c ##1 d) does not match, then evaluation of ts2 results in these two matches.

Sequence match items can be attached to the operand sequence of the **first_match** operator. The sequence match items are placed within the same set of parentheses that encloses the operand. Thus, for example, the local variable assignment x = e can be attached to the first match of seq via

```
first_match(seq, x = e)
```

which is equivalent to

```
first_match((seq, x = e))
```

See 17.8 and 17.9 for discussion of sequence match items.

17.7.8 Conditions over sequences

Sequences often occur under the assumptions of some conditions for correct behavior. A logical condition must hold true, for instance, while processing a transaction. Also, occurrence of certain values is prohibited while processing a transaction. Such situations can be expressed directly using the following construct:

sequence_expr ::=   *// from A.2.10*

    ...

    | expression_or_dist **throughout** sequence_expr

Syntax 17-10—Throughout construct syntax (excerpt from Annex A)

The construct exp **throughout** seq is an abbreviation for the following:

```
(exp) [*0:$] intersect seq
```

The composite sequence, exp **throughout** seq, matches along a finite interval of consecutive clock ticks provided seq matches along the interval and exp evaluates to true at each clock tick of the interval.

The following example is illustrated in Figure 17-11.

```
sequence burst_rule1;
   @(posedge mclk)
      $fell(burst_mode) ##0
      (!burst_mode) throughout (##2 ((trdy==0)&&(irdy==0)) [*7]);
endsequence
```
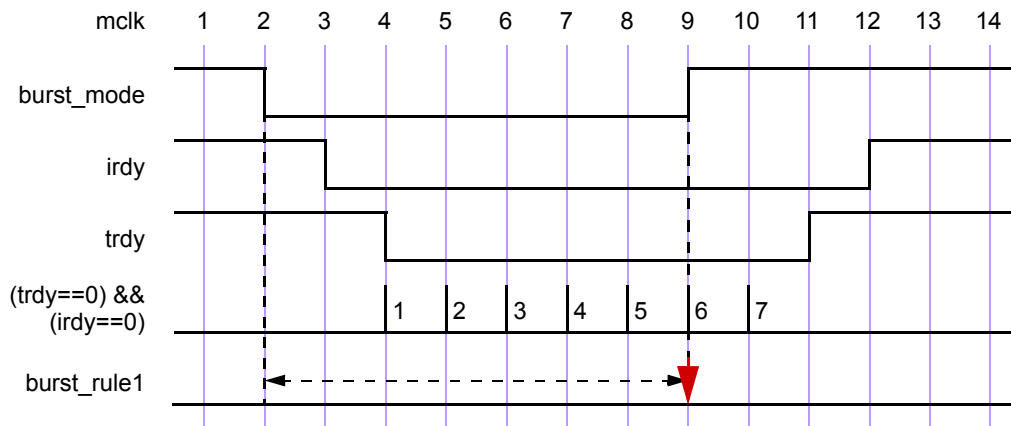

Figure 17-11—Match with throughout restriction fails

Figure 17-12 illustrates the evaluation attempt for sequence burst_rule1 beginning at clock tick 2. Because signal burst_mode is high at clock tick 1 and low at clock tick 2, $fell(burst_mode) is true at clock tick 2. To complete the match of burst_rule1, the value of burst_mode is required to be low throughout a match of the subsequence (##2 ((trdy==0)&&(irdy==0)) [*7]) beginning at clock tick 2.

This subsequence matches from clock tick 2 to clock tick 10. However, at clock tick 9 `burst_mode` becomes high, thereby failing to match according to the rules for **throughout**.

If signal `burst_mode` were instead to remain low through at least clock tick 10, then there would be a match of `burst_rule1` from clock tick 2 to clock tick 10, as shown in Figure 17-12.
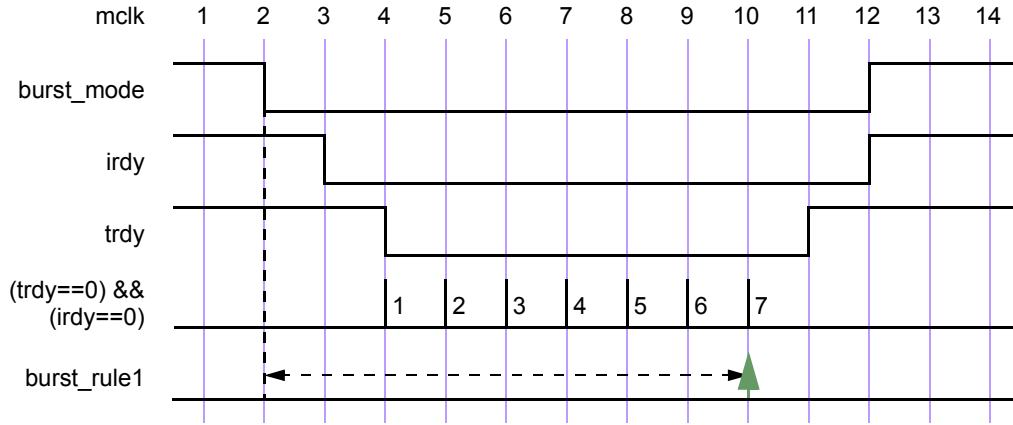
Figure 17-12—Match with throughout restriction succeeds

17.7.9 Sequence contained within another sequence

The containment of a sequence within another sequence is expressed as follows:

```
sequence_expr ::=                                                    // from A.2.10
    ...
    | sequence_expr within sequence_expr
```

Syntax 17-11—Within construct syntax (excerpt from Annex A)

The construct `seq1` **within** `seq2` is an abbreviation for the following:

    (1[*0:$] ##1 seq1 ##1 1[*0:$]) intersect seq2

The composite sequence `seq1` **within** `seq2` matches along a finite interval of consecutive clock ticks provided `seq2` matches along the interval and `seq1` matches along some subinterval of consecutive clock ticks. In other words, the matches of `seq1` and `seq2` must satisfy the following:

— The start point of the match of `seq1` must be no earlier than the start point of the match of `seq2`.

— The end point of the match of `seq1` must be no later than the end point of the match of `seq2`.

For example, the sequence

    !trdy[*7] within (($fell irdy) ##1 !irdy[*8])

matches from clock tick 3 to clock tick 11 on the trace shown in Figure 17-12.

17.7.10 Detecting and using end point of a sequence

There are two ways in which a complex sequence can be decomposed into simpler subsequences.

One is to instantiate a named sequence by referencing its name. Evaluation of such a reference requires the named sequence to match starting from the clock tick at which the reference is reached during the evaluation of the enclosing sequence. For example:

```
sequence s;
    a ##1 b ##1 c;
endsequence
sequence rule;
    @(posedge sysclk)
        trans ##1 start_trans ##1 s ##1 end_trans;
endsequence
```

Sequence `s` is evaluated beginning one tick after the evaluation of `start_trans` in the sequence `rule`.

Another way to use a sequence is to detect its end point in another sequence. The end point of a sequence is reached whenever the ending clock tick of a match of the sequence is reached, regardless of the starting clock tick of the match. The reaching of the end point can be tested by using the method `ended`.

The syntax of the ended method is as follows:

```
sequence_instance.ended
```

`ended` is a method on a sequence. The result of its operation is true or false. When method `ended` is evaluated in an expression, it tests whether its operand sequence has reached its end point at that particular point in time. The result of `ended` does not depend upon the starting point of the match of its operand sequence. An example is shown below:

```
sequence e1;
    @(posedge sysclk) $rose(ready) ##1 proc1 ##1 proc2 ;
endsequence
sequence rule;
    @(posedge sysclk) reset ##1 inst ##1 e1.ended ##1 branch_back;
endsequence
```

In this example, sequence `e1` must match one clock tick after `inst`. If the method `ended` is replaced with an instance of sequence `e1`, a match of `e1` must start one clock tick after `inst`. Notice that method `ended` only tests for the end point of `e1` and has no bearing on the starting point of `e1`. `ended` can be used on sequences that have formal arguments. For example, with the declarations

```
sequence e2(a,b,c);
    @(posedge sysclk) $rose(a) ##1 b ##1 c;
endsequence
sequence rule2;
    @(posedge sysclk) reset ##1 inst ##1 e2(ready,proc1,proc2).ended
        ##1 branch_back;
endsequence
```

`rule2` is equivalent to `rule2a` below:

```
sequence e2_instantiated;
  e2(ready,proc1,proc2);
endsequence
sequence rule2a;
  @(posedge sysclk) reset ##1 inst ##1 e2_instantiated.ended ##1 branch_back;
endsequence
```

There are additional restrictions on passing local variables into an instance of a sequence to which `ended` is applied. See 17.8.

Method `ended` can be used in the presence of multiple clocks. However, the ending clock of the sequence instance to which `ended` is applied must always be the same as the clock in the context where the application of method `ended` appears. See 17.12.5.

17.8 Manipulating data in a sequence

The use of a static SystemVerilog variable implies that only one copy exists. If data values need to be checked in pipelined designs, then for each quantum of data entering the pipeline, a separate variable can be used to store the predicted output of the pipeline for later comparison when the result actually exits the pipe. This storage can be built by using an array of variables arranged in a shift register to mimic the data propagating through the pipeline. However, in more complex situations where the latency of the pipe is variable and out of order, this construction could become very complex and error prone. Therefore, variables are needed that are local to and are used within a particular transaction check that can span an arbitrary interval of time and can overlap with other transaction checks. Such a variable must thus be dynamically created when needed within an instance of a sequence and removed when the end of the sequence is reached.

The dynamic creation of a variable and its assignment is achieved by using the local variable declaration in a sequence or property declaration and making an assignment in the sequence.

```
sequence_expr ::=                                                    // from A.2.10
      ...
     | ( expression_or_dist {, sequence_match_item } ) [ boolean_abbrev ]
     | ( sequence_expr {, sequence_match_item} ) [ sequence_abbrev ]
      ...
sequence_match_item ::=
      operator_assignment
    | inc_or_dec_expression
    | subroutine_call
```

Syntax 17-12—Variable assignment syntax (excerpt from Annex A)

The type of variable is explicitly specified. The variable can be assigned at the end point of any syntactic subsequence by placing the subsequence, comma separated from the sampling assignment, in parentheses. For example, if in

```
    a ##1 b[->1] ##1 c[*2]
```

it is desired to assign `x = e` at the match of `b[->1]`, the sequence can be rewritten as

```
    a ##1 (b[->1], x = e) ##1 c[*2]
```

The local variable can be reassigned later in the sequence, as in

```
    a ##1 (b[->1], x = e) ##1 (c[*2], x = x + 1)
```

For every attempt, a new copy of the variable is created for the sequence. The variable value can be tested like any other SystemVerilog variable.

Hierarchical references to a local variable are not allowed.

As an example of local variable usage, assume a pipeline that has a fixed latency of five clock cycles. The data enter the pipe on `pipe_in` when `valid_in` is true, and the value computed by the pipeline appears five clock cycles later on the signal `pipe_out1`. The data as transformed by the pipe are predicted by a function that increments the data. The following property verifies this behavior:

```
property e;
    int x;
    (valid_in, x = pipe_in) |-> ##5 (pipe_out1 == (x+1));
endproperty
```

Property `e` is evaluated as follows:

— When `valid_in` is true, `x` is assigned the value of `pipe_in`. If five cycles later, `pipe_out1` is equal to `x+1`, then property `e` is true. Otherwise, property `e` is false.

— When is `valid_in` false, property `e` evaluates to true.

A local variable can be used to form expressions in the same way that a static variable of the same type can be used. This includes the use of local variables in expressions for bit-selects and part-selects of vectors or for indexes of arrays.

Local variables can be used in sequences or properties.

```
sequence data_check;
    int x;
    a ##1 (!a, x = data_in) ##1 !b[*0:$] ##1 b && (data_out == x);
endsequence
property data_check_p
    int x;
    a ##1 (!a, x = data_in) |=> !b[*0:$] ##1 b && (data_out == x);
endproperty
```

Local variables can be written on repeated sequences and accomplish accumulation of values.

```
sequence rep_v;
    int x;
    'true,x = 0 ##0
    (!a [* 0:$] ##1 a, x = x+data)[*4] ##1 b ##1 c && (data_out == x);
endsequence
```

The local variables declared in one sequence are not visible in the sequence where it gets instantiated. The example below illustrates an illegal access to local variable `v1` of sequence `sub_seq1` in sequence `seq1`.

```
sequence sub_seq1;
    int v1;
    (a ##1 !a, v1 = data_in) ##1 !b[*0:$] ##1 b && (data_out == v1);
endsequence
sequence seq1;
    c ##1 sub_seq1 ##1 (do1 == v1); // error because v1 is not visible
endsequence
```

To access a local variable of a subsequence, a local variable must be declared and passed to the instantiated subsequence through an argument. The example below illustrates this usage.

```
sequence sub_seq2(lv);
    (a ##1 !a, lv = data_in) ##1 !b[*0:$] ##1 b && (data_out == lv);
endsequence
sequence seq2;
```

```
    int v1;
    c ##1 sub_seq2(v1) ##1 (do1 == v1); // v1 is now bound to lv
endsequence
```

Local variables can be passed into an instance of a named sequence to which ended is applied and accessed in a similar manner. For example:

```
sequence seq2a;
    int v1; c ##1 sub_seq2(v1).ended ##1 (do1 == v1); // v1 is now bound to lv
endsequence
```

There are additional restrictions when passing local variables into an instance of a named sequence to which ended is applied:

— Local variables can be passed in only as entire actual arguments, not as proper subexpressions of actual arguments.

— In the declaration of the named sequence, the formal argument to which the local variable is bound must not be referenced before it is assigned.

The second restriction is met by sub_seq2 because the assignment lv = data_in occurs before the reference to lv in data_out == lv.

If a local variable is assigned before being passed into an instance of a named sequence to which ended is applied, then the restrictions prevent this assigned value from being visible within the named sequence. The restrictions are important because the use of ended means that there is no guaranteed relationship between the point in time at which the local variable is assigned outside the named sequence and the beginning of the match of the instance.

A local variable that is passed in as actual argument to an instance of a named sequence to which ended is applied will flow out of the application of ended to that instance provided both of the following conditions are met:

— The local variable flows out of the end of the named sequence instance, as defined by the local variable flow rules for sequences. (See below and Annex E.)

— The application of ended to this instance is a maximal boolean expression. In other words, the application of ended cannot have negation or any other expression operator applied to it.

Both conditions are satisfied by sub_seq2 and seq2a. Thus, in seq2a, the value in v1 in the comparison do1 == v1 is the value assigned to lv in sub_seq2 by the assignment lv = data_in. However, in

```
sequence seq2b;
    int v1; c ##1 !sub_seq2(v1).ended ##1 (do1 == v1); // v1 unassigned
endsequence
```

the second condition is violated because of the negation applied to sub_seq2(v1).ended. Therefore, v1 does not flow out of the application of ended to this instance, and the reference to v1 in do1 == v1 is to an unassigned variable.

In a single cycle, there can be multiple matches of a sequence instance to which ended is applied, and these matches can have different valuations of the local variables. The multiple matches are treated semantically the same way as matching both disjuncts of an **or** (see below). In other words, the thread evaluating the instance to which ended is applied will fork to account for such distinct local variable valuations.

When a local variable is a formal argument of a sequence declaration, it is illegal to declare the variable, as shown below.

```
sequence sub_seq3(lv);
   int lv; // illegal because lv is a formal argument
   (a ##1 !a, lv = data_in) ##1 !b[*0:$] ##1 b && (data_out == lv);
endsequence
```

There are special considerations when using local variables in sequences involving the branching operators **or**, **and**, and **intersect**. The evaluation of a composite sequence constructed from one of these operators can be thought of as forking two threads to evaluate the operand sequences in parallel. A local variable may have been assigned a value before the start of the evaluation of the composite sequence. Such a local variable is said to flow in to each of the operand sequences. The local variable may be assigned or reassigned in one or both of the operand sequences. In general, there is no guarantee that evaluation of the two threads results in consistent values for the local variable, or even that there is a consistent view of whether the local variable has been assigned a value. Therefore, the values assigned to the local variable before and during the evaluation of the composite sequence are not always allowed to be visible after the evaluation of the composite sequence.

In some cases, inconsistency in the view of the local variable's value does not matter, while in others it does. Precise conditions are given in Annex E to define static (i.e., compile-time computable) conditions under which a sufficiently consistent view of the local variable's value after the evaluation of the composite sequence is guaranteed. If these conditions are satisfied, then the local variable is said to flow out of the composite sequence. An intuitive description of the conditions for local variable flow follows:

a) Variables assigned on parallel threads cannot be accessed in sibling threads. For example:

```
sequence s4;
   int x;
   (a ##1 (b, x = data) ##1 c) or (d ##1 (e==x)); // illegal
endsequence
```

b) In the case of **or**, a local variable flows out of the composite sequence if, and only if, it flows out of each of the operand sequences. If the local variable is not assigned before the start of the composite sequence and it is assigned in only one of the operand sequences, then it does not flow out of the composite sequence.

c) Each thread for an operand of an **or** that matches its operand sequence continues as a separate thread, carrying with it its own latest assignments to the local variables that flow out of the composite sequence. These threads do not have to have consistent valuations for the local variables. For example:

```
sequence s5;
   int x,y;
   ((a ##1 (b, x = data, y = data1) ##1 c)
      or (d ##1 (`true, x = data) ##0 (e==x))) ##1 (y==data2);
   // illegal because y is not in the intersection
endsequence
sequence s6;
   int x,y;
   ((a ##1 (b, x = data, y = data1) ##1 c)
      or (d ##1 (`true, x = data) ##0 (e==x))) ##1 (x==data2);
   // legal because x is in the intersection
endsequence
```

d) In the case of **and** and **intersect**, a local variable that flows out of at least one operand shall flow out of the composite sequence unless it is blocked. A local variable is blocked from flowing out of the composite sequence if either of the following statements applies:

1) The local variable is assigned in and flows out of each operand of the composite sequence, or

2) The local variable is blocked from flowing out of at least one of the operand sequences.

The value of a local variable that flows out of the composite sequence is the latest assigned value. The threads for the two operands are merged into one at completion of evaluation of the composite sequence.

```
sequence s7;
   int x,y;
   ((a ##1 (b, x = data, y = data1) ##1 c)
      and (d ##1 ('true, x = data) ##0 (e==x))) ##1 (x==data2);
   // illegal because x is common to both threads
endsequence
sequence s8;
   int x,y;
   (a ##1 (b, x = data, y = data1) ##1 c)
      and (d ##1 ('true, x = data) ##0 (e==x))) ##1 (y==data2);
   // legal because y is in the difference
endsequence
```

17.9 Calling subroutines on match of a sequence

Tasks, task methods, void functions, void function methods, and system tasks can be called at the end of a successful match of a sequence. The subroutine calls, like local variable assignments, appear in the comma-separated list that follows the sequence. The subroutine calls are said to be attached to the sequence. The sequence and the list that follows are enclosed in parentheses.

```
sequence_expr ::=                                                   // from A.2.10
     ...
   | ( expression_or_dist {, sequence_match_item } ) [ boolean_abbrev ]
   | ( sequence_expr {, sequence_match_item} ) [ sequence_abbrev ]
     ...
sequence_match_item ::=
     operator_assignment
   | inc_or_dec_expression
   | subroutine_call
```

Syntax 17-13—Subroutine call in sequence syntax (excerpt from *Annex A*)

For example:

```
sequence s1;
   logic v, w;
   (a, v = e) ##1
   (b[->1], w = f, $display("b after a with v = %h, w = %h\n", v, w));
endsequence
```

defines a sequence s1 that matches at the first occurrence of b strictly after an occurrence of a. At the match, the system task $display is executed to write a message that announces the match and shows the values assigned to the local variables v and w.

All subroutine calls attached to a sequence are executed at every successful match of the sequence. For each successful match, the attached calls are executed in the order they appear in the list. The subroutines are scheduled in the Reactive region, like an action block.

Each argument of a subroutine call attached to a sequence must either be passed by value as an input or be passed by reference (either **ref** or **const ref**; see 12.4.2). Actual argument expressions that are passed by

value use sampled values of the underlying variables and are consistent with the variable values used to evaluate the sequence match.

Local variables can be passed into subroutine calls attached to a sequence. Any local variable that flows out of the sequence or that is assigned in the list following the sequence, but before the subroutine call, can be used in an actual argument expression for the call. If a local variable appears in an actual argument expression, then that argument must be passed by value.

17.10 System functions

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is "one-hot". The following system functions are included to facilitate such common assertion functionality:

— `$onehot (<expression>)` returns true if only 1 bit of the expression is high.
— `$onehot0(<expression>)` returns true if at most 1 bit of the expression is high.
— `$isunknown  (<expression>)` returns true if any bit of the expression is X or Z. This is equivalent to `^<expression> === 'bx`.

All of the above system functions have a return type of **bit**. A return value of `1'b1` indicates true, and a return value of `1'b0` indicates false.

Another useful function provided for the boolean expression is `$countones`, to count the number of ones in a bit vector expression.

```
$countones ( expression)
```

An X and Z value of a bit is not counted towards the number of ones.

17.11 Declaring properties

A property defines a behavior of the design. A property can be used for verification as an assumption, a checker, or a coverage specification. In order to use the behavior for verification, an **assert**, **assume**, or **cover** statement must be used. A property declaration by itself does not produce any result.

A **property** can be declared in any of the following:
— A module
— An interface
— A program
— A **clocking** block
— A package
— A compilation-unit scope

To declare a property, the **property** construct is used as shown below:

```
concurrent_assertion_item_declaration ::=                                    // from A.2.10
        property_declaration
        ...
property_declaration ::=
        property property_identifier [ ( [ tf_port_list ] ) ] ;
            { assertion_variable_declaration }
            property_spec ;
        endproperty [ : property_identifier ]
list_of_formals ::= formal_list_item { , formal_list_item }
property_spec ::=
        [clocking_event ] [ disable iff ( expression_or_dist ) ] property_expr
property_expr ::=
        sequence_expr
    | ( property_expr )
    | not property_expr
    | property_expr or property_expr
    | property_expr and property_expr
    | sequence_expr |-> property_expr
    | sequence_expr |=> property_expr
    | if ( expression_or_dist ) property_expr [ else property_expr ]
    | property_instance
    | clocking_event property_expr
assertion_variable_declaration ::=
        var_data_type list_of_variable_identifiers ;
property_instance::=
        ps_property_identifier [ ( [ list_of_arguments ] ) ]
```

Syntax 17-14—Property construct syntax (excerpt from Annex A)

A **property** is declared with optional formal arguments, as in a sequence declaration. When a property is instantiated, actual arguments can be passed to the property. The mechanism for passing arguments to a property is the same as for passing arguments to a sequence. The property gets expanded with the actual arguments by replacing the formal arguments with the actual arguments. Semantic checks are performed to ensure that the expanded property with the actual arguments is legal.

The result of property evaluation is either true or false. There are seven kinds of property: sequence, negation, disjunction, conjunction, **if**...**else**, implication, and instantiation.

a) A property that is a sequence evaluates to true if, and only if, there is a nonempty match of the sequence. A sequence that admits an empty match is not allowed as a property. Because there is a match if, and only if, there is a first match, evaluation of such a property is the same as implicitly transforming its *sequence_expr* to `first_match(sequence_expr)`. As soon as a match of *sequence_expr* is determined, the evaluation of the property is considered to be true, and no other matches are required for that evaluation attempt.

b) A property is a negation if it has the form

    not property_expr

    For each evaluation attempt of the property, there is an evaluation attempt of *property_expr*. The keyword **not** states that the evaluation of the property returns the opposite of the evaluation of the

underlying *property_expr*. Thus, if *property_expr* evaluates to true, then **not** *property_expr* evaluates to false; and if *property_expr* evaluates to false, then **not** *property_expr* evaluates to true.

c) A property is a disjunction if it has the form

```
property_expr1 or property_expr2
```

The property evaluates to true if, and only if, at least one of `property_expr1` and `property_expr2` evaluates to true.

d) A property is a conjunction if it has the form

```
property_expr1 and property_expr2
```

The property evaluates to true if, and only if, both `property_expr1` and `property_expr2` evaluate to true.

e) A property is an **if**...**else** if it has either the form

```
if (expression_or_dist) property_expr1
```

or the form

```
if (expression_or_dist) property_expr1 else property_expr2
```

A property of the first form evaluates to true if, and only if, either *expression_or_dist* evaluates to false or `property_expr1` evaluates to true. A property of the second form evaluates to true if, and only if, either *expression_or_dist* evaluates to true and `property_expr1` evaluates to true or *expression_or_dist* evaluates to false and `property_expr2` evaluates to true.

f) A property is an implication if it has either the form

```
sequence_expr |-> property_expr
```

or the form

```
sequence_expr |=> property_expr
```

The meaning of implications is discussed in 17.11.2.

g) An instance of a named property can be used as a *property_expr* or *property_spec*. In general, the instance is legal provided the body *property_spec* of the named property can be substituted in place of the instance, with actual arguments substituted for formal arguments, and result in a legal *property_expr* or *property_spec*, ignoring local variable declarations. Thus, for example, if an instance of a named property is used as a *property_expr* operand for any property-building operator, then the named property must not have a **disable iff** clause. Similarly, clock events in a named property must conform to the rules of multiclock support when the property is instantiated in a *property_expr* or *property_spec* that also involves other clock events.

Table 17-2 lists the sequence and property operators from highest to lowest precedence and shows the associativity of the non-unary operators.

Table 17-2—Sequence and property operator precedence and associativity

Sequence operators	Property operators	Associativity
`[*]`, `[=]`, `[->]`		—
`##`		Left
throughout		Right
within		Left
intersect		Left
	not	—
and	**and**	Left
or	**or**	Left
	if…**else**	Right
	`|->`, `|=>`	Right

A **disable iff** clause can be attached to a *property_expr* to yield a *property_spec*.

```
disable iff (expression_or_dist) property_expr
```

The expression of the **disable iff** is called the *reset expression*. The **disable iff** clause allows preemptive resets to be specified. For an evaluation of the *property_spec*, there is an evaluation of the underlying *property_expr*. If prior to the completion of that evaluation the reset expression becomes true, then the overall evaluation of the *property_spec* is true. Otherwise, the evaluation of the *property_spec* is the same as that of the *property_expr*. The reset expression is tested independently for different evaluation attempts of the *property_spec*. The values of variables used in the reset expression are those in the current simulation cycle, i.e., not sampled. The expression may contain a reference to an end point of a sequence by using the method *triggered* of that sequence. *Matched* and *ended* of a sequence and local variables cannot be used in the reset expression. If a sampled value function is used in the reset expression, the sampling clock must be explicitly specified in its actual argument list as described in 17.7.3. Nesting of **disable iff** clauses, explicitly or through property instantiations, is not allowed.

17.11.1 Typed formal arguments in property declarations

Formal arguments of properties can optionally be typed. To declare a type for a formal argument of a property, it is required to prefix the argument with a type. A formal argument that is not prefixed by a type shall be untyped.

The supported data types for property formal arguments are the types that are allowed for operands in assertion expressions (see 17.4.1). The assignment rules for assigning actual arguments to formal arguments, at the time of property instantiation, are the same as the general rules for doing assignment of a typed variable with another typed expression (see Clause 4).

For example, below are two equivalent ways of passing arguments. The first has untyped arguments, and the second has typed arguments:

```
property rule6_with_no_type(x, y);
    ##1 x |-> ##[2:10] y;
endproperty
```

```
property rule6_with_type(bit x, bit y);
    ##1 x |-> ##[2:10] y;
endproperty
```

17.11.2 Implication

The implication construct specifies that the checking of a property is performed conditionally on the match of a sequential antecedent.

property_expr ::=   *// from A.2.10*

   ...
   | sequence_expr **|->** property_expr
   | sequence_expr **|=>** property_expr

Syntax 17-15—Implication syntax (excerpt from Annex A)

This clause is used to precondition monitoring of a property expression and is allowed at the property level. The result of the implication is either true or false. The left-hand operand *sequence_expr* is called the *antecedent*, while the right-hand operand *property_expr* is called the *consequent*.

The following points should be noted for `|->` implication:

— From a given start point, the antecedent *sequence_expr* can have zero, one, or more than one successful match.

— If there is no match of the antecedent *sequence_expr* from a given start point, then evaluation of the implication from that start point succeeds vacuously and returns true.

— For each successful match of antecedent *sequence_expr*, the consequent *property_expr* is separately evaluated. The end point of the match of the antecedent *sequence_expr* is the start point of the evaluation of the consequent *property_expr*.

— From a given start point, evaluation of the implication succeeds and returns true if, and only if, for every match of the antecedent *sequence_expr* beginning at the start point, the evaluation of the consequent *property_expr* beginning at the end point of the match succeeds and returns true.

Two forms of implication are provided: overlapped using operator `|->` and nonoverlapped using operator `|=>`. For overlapped implication, if there is a match for the antecedent *sequence_expr*, then the end point of the match is the start point of the evaluation of the consequent *property_expr*. For nonoverlapped implication, the start point of the evaluation of the consequent *property_expr* is the clock tick after the end point of the match. Therefore,

```
sequence_expr |=> property_expr
```

is equivalent to the following:

```
sequence_expr ##1 'true |-> property_expr
```

The use of implication when multiclock sequences and properties are involved is explained in 17.12.

The following example illustrates a bus operation for data transfer from a master to a target device. When the bus enters a data transfer phase, multiple data phases can occur to transfer a block of data. During the data transfer phase, a data phase completes on any rising clock edge on which `irdy` is asserted and either `trdy` or `stop` is asserted. In this example, an asserted signal implies a value of low. The end of a data phase can be expressed as follows:

```
property data_end;
```

```
    @(posedge mclk)
    data_phase |-> ((irdy==0) && ($fell(trdy) || $fell(stop))) ;
endproperty
```

Each time a data phase is true, a match for `data_phase` is recognized. The attempt at clock tick 6 is illustrated in Figure 17-13. The values shown for the signals are the sampled values with respect to the clock. At clock tick 6, `data_end` is true because `stop` gets asserted while `irdy` is asserted.
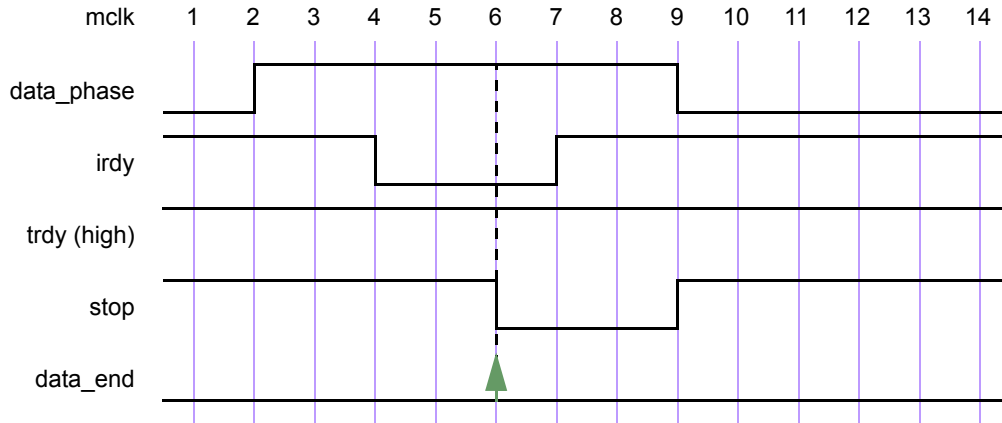
Figure 17-13—Conditional sequence matching

In another example, `data_end_exp` is used to ensure that `frame` is deasserted (value high) within two clock ticks after `data_end_exp` occurs. Further, it is also required that `irdy` is deasserted (value high) one clock tick after `frame` is deasserted.

A property written to express this condition is shown below.

```
`define data_end_exp (data_phase && ((irdy==0)&&($fell(trdy)||$fell(stop))))
property data_end_rule1;
    @(posedge mclk)
    `data_end_exp |-> ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty
```

Property `data_end_rule1` first evaluates `data_end_exp` at every clock tick to test if its value is true. If the value is false, then that particular attempt to evaluate `data_end_rule1` is considered true. Otherwise, the following sequence is evaluated:

```
##[1:2] $rose(frame) ##1 $rose(irdy)
```

specifies looking for the rising edge of `frame` within two clock ticks in the future. After `frame` toggles high, `irdy` must also toggle high after one clock tick. This is illustrated in Figure 17-14 for the evaluation attempt at clock tick 6. `data_end_exp` is acknowledged at clock tick 6. Next, `frame` toggles high at clock tick 7. Because this falls within the timing constraint imposed by `[1:2]`, it satisfies the sequence and continues to evaluate further. At clock tick 8, `irdy` is evaluated. Signal `irdy` transitions to high at clock tick 8, matching the sequence specification completely for the attempt that began at clock tick 6.
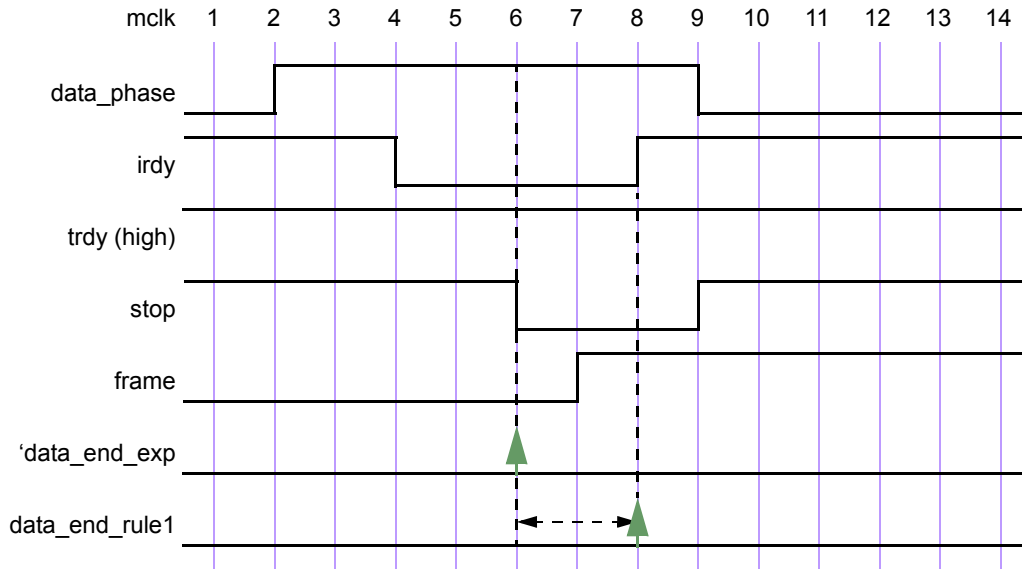
Figure 17-14—Conditional sequences

Generally, assertions are associated with preconditions so that the checking is performed only under certain specified conditions. As seen from the previous example, the |-> operator provides this capability to specify preconditions with sequences that must be satisfied before evaluating their consequent properties. The next example modifies the preceding example to see the effect on the results of the assertion by removing the pre-condition for the consequent. This is shown below and illustrated in Figure 17-15.

```
property data_end_rule2;
    @(posedge mclk) ##[1:2] $rose(frame) ##1 $rose(irdy);
endproperty
```
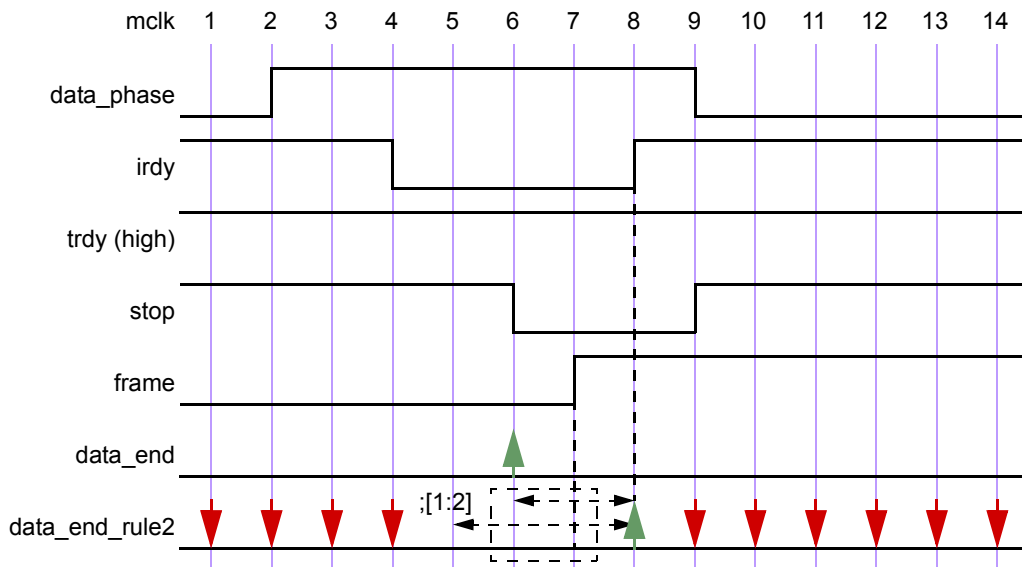

Figure 17-15—Results without the condition

The property is evaluated at every clock tick. For the evaluation at clock tick 1, the rising edge of signal frame does not occur at clock tick 2 or 3; therefore, the property fails at clock tick 1. Similarly, there is a failure at clock ticks 2, 3, and 4. For attempts starting at clock ticks 5 and 6, the rising edge of signal frame at clock tick 7 allows checking further. At clock tick 8, the sequences complete according to the specification, resulting in a match for attempts starting at clock ticks 5 and 6. All later attempts to match the sequence fail because $rose(frame) does not occur again.

Figure 17-15 shows that removing the precondition of checking `data_end_exp from the assertion causes failures that are not relevant to the verification objective. It is important from the validation standpoint to determine these preconditions and use them to filter out inappropriate or extraneous situations.

An example of implication where the antecedent is a sequence follows:

```
(a ##1 b ##1 c) |-> (d ##1 e)
```

If the sequence (a ##1 b ##1 c) matches, then the sequence (d ##1 e) must also match. On the other hand, if the sequence (a ##1 b ##1 c) does not match, then the result is true.

Another example of implication is as follows:

```
property p16;
    (write_en & data_valid) ##0
    (write_en && (retire_address[0:4]==addr)) [*2] |->
    ##[3:8] write_en && !data_valid &&(write_address[0:4]==addr);
endproperty
```

This property can be coded alternatively as a nested implication:

```
property p16_nested;
    (write_en & data_valid) |->
        (write_en && (retire_address[0:4]==addr)) [*2] |->
            ##[3:8] write_en && !data_valid && (write_address[0:4]==addr);
endproperty
```

Multiclock sequence implication is explained in 17.12.

17.11.3 Property examples

The following examples illustrate the property forms:

```
property rule1;
    @(posedge clk) a |-> b ##1 c ##1 d;
endproperty
property rule2;
    @(clkev) disable iff (foo) a |-> not(b ##1 c ##1 d);
endproperty
```

Property rule2 negates the sequence (b ##1 c ##1 d) in the consequent of the implication. clkev specifies the clock for the property.

```
property rule3;
    @(posedge clk) a[*2] |-> ((##[1:3] c) or (d |=> e));
endproperty
```

Property rule3 says that if a holds and a also held last cycle, then either c must hold at some point one to three cycles after the current cycle or, if d holds in the current cycle, then e must hold one cycle later.

```
property rule4;
    @(posedge clk) a[*2] |-> ((##[1:3] c) and (d |=> e));
endproperty
```

Property `rule4` says that if `a` holds and `a` also held last cycle, then `c` must hold at some point one to three cycles after the current cycle and, if `d` holds in the current cycle, then `e` must hold one cycle later.

```
property rule5;
    @(posedge clk)
    a ##1 (b || c)[->1] |->
        if (b)
            (##1 d |-> e)
        else // c
            f ;
endproperty
```

Property `rule5` has `a` followed by the next occurrence of either `b` or `c` as its antecedent. The consequent uses **if**...**else** to split cases on which of `b` or `c` is matched first.

```
property rule6(x,y);
    ##1 x |-> y;
endproperty
property rule5a;
    @(posedge clk)
    a ##1 (b || c)[->1] |->
        if (b)
            rule6(d,e)
        else // c
            f ;
endproperty
```

Property `rule5a` is equivalent to `rule5`, but it uses an instance of `rule6` as a property expression.

A property can optionally specify an event control for the clock. The clock derivation and resolution rules are described in 17.14.

A named property can be instantiated by referencing its name. A hierarchical name can be used, consistent with the SystemVerilog naming conventions. Like sequence declarations, variables used within a property that are not formal arguments to the property are resolved hierarchically from the scope in which the property is declared.

Properties that use more than one clock are described in 17.12.

17.11.4 Recursive properties

SystemVerilog allows recursive properties. A named property is recursive if its declaration involves an instantiation of itself. Recursion provides a flexible framework for coding properties to serve as ongoing assumptions, checkers, or coverage monitors.

For example:

```
property prop_always(p);
    p and (1'b1 |=> prop_always(p));
endproperty
```

is a recursive property that says that the formal argument property `p` must hold at every cycle. This example is useful if the ongoing requirement that property `p` hold applies after a complicated triggering condition encoded in sequence `s`:

```
property p1(s,p);
    s |=> prop_always(p);
endproperty
```

As another example, the recursive property

```
property prop_weak_until(p,q);
    q or (p and (1'b1 |=> prop_weak_until(p,q)));
endproperty
```

says that formal argument property `p` must hold at every cycle up to, but not including, the first cycle at which formal argument property `q` holds. Formal argument property `q` is not required ever to hold, however. This example is useful if `p` must hold at every cycle after a complicated triggering condition encoded in sequence `s`, but the requirement on `p` is lifted by `q`:

```
property p2(s,p,q);
    s |=> prop_weak_until(p,q);
endproperty
```

More generally, several properties can be mutually recursive. For example:

```
property check_phase1;
    s1 |-> (phase1_prop and (1'b1 |=> check_phase2));
endproperty
property check_phase2;
    s2 |-> (phase2_prop and (1'b1 |=> check_phase1));
endproperty
```

There are four restrictions on recursive property declarations.

— RESTRICTION 1: The negation operator **not** cannot be applied to any property expression that instantiates a recursive property. In particular, the negation of a recursive property cannot be asserted or used in defining another property.

Here are examples of illegal property declarations that violate Restriction 1:

```
property illegal_recursion_1(p);
    not prop_always(not p);
endproperty

property illegal_recursion_2(p);
    p and (1'b1 |=> not illegal_recursion_2(p));
endproperty
```

Furthermore, **not** cannot be applied to any property expression that instantiates a property that depends on a recursive property. The precise definition of dependency is given in Annex E.

— RESTRICTION 2: The operator **disable iff** cannot be used in the declaration of a recursive property. This restriction is consistent with the restriction that **disable iff** cannot be nested.

Below is an example of an illegal property declaration that violates Restriction 2:

```
property illegal_recursion_3(p);
    disable iff (b)
    p and (1'b1 |=> illegal_recursion_3(p));
```

```
        endproperty
```

The intent of `illegal_recursion_3` can be written legally as

```
    property legal_3(p);
        disable iff (b) prop_always(p);
    endproperty
```

because `legal_3` is not a recursive property.

— RESTRICTION 3: If `p` is a recursive property, then, in the declaration of `p`, every instance of `p` must occur after a positive advance in time. In the case of mutually recursive properties, all recursive instances must occur after positive advances in time.

Below is an example of an illegal property declaration that violates Restriction 3:

```
    property illegal_recursion_4(p);
        p and (1'b1 |-> illegal_recursion_4(p));
    endproperty
```

If this form were legal, the recursion would be stuck in time, checking `p` over and over again at the same cycle.

— RESTRICTION 4: For every recursive instance of property *q* in the declaration of property *p*, each actual argument expression e of the instance satisfies at least one of the following conditions:

— e is itself a formal argument of `p`.

— No formal argument of `p` appears in e.

— e is passed to a formal argument of `q` that is typed and the set of values for the type is bounded.

For example:

```
    property p1(int i, j);
        (j == 7) or
        (
            (i > 0) and
            ((a ##1 b) |-> p1(i+2, j))
        );
    endproperty
```

is a legal declaration, but

```
    property p1(int i, j);
        (j == 7) or
        (
            (i > 0) and
            ((a ##1 b) |-> p1(i+2, j+2))
        );
    endproperty
```

is not legal because the second formal argument of `p1` is not typed, the actual argument expression `j+2` in the recursive instance `p1(i+2, j+2)` is not itself a formal argument of `p1`, and this actual argument expression has an appearance of the formal argument `j` of `p1`.

Recursive properties can represent complicated requirements, such as those associated with varying numbers of data beats, out-of-order completions, retries, etc. Below is an example of using a recursive property to check complicated conditions of this kind.

For example, suppose that write data must be checked according to the following conditions:

— Acknowledgment of a write request is indicated by the signal `write_request` together with `write_request_ack`. When a write request is acknowledged, it gets a 4-bit tag, indicated by signal `write_reqest_ack_tag`. The tag is used to distinguish data beats for multiple write transactions in flight at the same time.

— It is understood that distinct write transactions in flight at the same time must be given distinct tags. For simplicity, this condition is not a part of what is checked in this example.

— Each write transaction can have between 1 and 16 data beats, and each data beat is 8 bits. There is a model of the expected write data that is available at acknowledgment of a write request. The model is a 128-bit vector. The most significant group of 8 bits represents the expected data for the first beat, the next group of 8 bits represents the expected data for the second beat (if there is a second beat), and so forth.

— Data transfer for a write transaction occurs after acknowledgment of the write request and, barring retry, ends with the last data beat. The data beats for a single write transaction occur in order.

— A data beat is indicated by the `data_valid` signal together with the signal `data_valid_tag` to determine the relevant write transaction. The signal data are valid with `data_valid` and carry the data for that beat. The data for each beat must be correct according to the model of the expected write data.

— The last data beat is indicated by signal `last_data_valid` together with `data_valid` and `data_valid_tag`. For simplicity, this example does not represent the number of data beats and does not check that `last_data_valid` is signaled at the correct beat.

— At any time after acknowledgment of the write request, but not later than the cycle after the last data beat, a write transaction can be forced to retry. Retry is indicated by the signal `retry` together with signal `retry_tag` to identify the relevant write transaction. If a write transaction is forced to retry, then its current data transfer is aborted, and the entire data transfer must be repeated. The transaction does not re-request, and its tag does not change.

— There is no limit on the number of times a write transaction can be forced to retry.

— A write transaction completes the cycle after the last data beat provided it is not forced to retry in that cycle.

Here is code to check these conditions:

```
property check_write;

  logic [0:127] expected_data;  // local variable to sample model data
  logic [3:0] tag;              // local variable to sample tag

  disable iff (reset)
  (
     write_request && write_request_ack,
     expected_data = model_data,
     tag = write_request_ack_tag
  )
  |=>
  check_write_data_beat(expected_data, tag, 4'h0);

endproperty

property check_write_data_beat
(
   expected_data,  // [0:127]
   tag,            // [3:0]
   i               // [3:0]
```

```
      );

          first_match
          (
              ##[0:$]
              (
                  (data_valid && (data_valid_tag == tag))
                  ||
                  (retry && (retry_tag == tag))
              )
          )
          |->
          (
              (
                  (data_valid && (data_valid_tag == tag))
                  |->
                  (data == expected_data[i*8+:8])
              )
              and
              (
                  if (retry && (retry_tag == tag))
                  (
                      1'b1 |=> check_write_data_beat(tag, expected_data, 4'h0)
                  )
                  else if (!last_data_valid)
                  (
                      1'b1 |=> check_write_data_beat(tag, expected_data, i+4'h1)
                  )
                  else
                  (
                      ##1 (retry && (retry_tag == tag))
                      |=>
                      check_write_data_beat(tag, expected_data, 4'h0)
                  )
              )
          );

      endproperty
```

17.11.5 Finite-length versus infinite-length behavior

The formal semantics in [Annex E](#) defines whether a given property holds on a given behavior. How the outcome of this evaluation relates to the design depends on the behavior that was analyzed. In dynamic verification, only behaviors that are finite in length are considered. In such a case, SystemVerilog defines four levels of satisfaction of a property:

— Holds strongly

    — No bad states have been seen.

    — All future obligations have been met.

    — The property will hold on any extension of the path.

— Holds (but does not hold strongly)

    — No bad states have been seen.

    — All future obligations have been met.

    — The property may or may not hold on a given extension of the path.

— Pending

    — No bad states have been seen.

    — Future obligations have not been met.

    — The property may or may not hold on a given extension of the path.

— Fails

    — A bad state has been seen.

    — Future obligations may or may not have been met.

    — The property will not hold on any extension of the path.

17.11.6 Nondegeneracy

It is possible to define sequences that can never be matched. For example:

```
(1'b1) intersect(1'b1 ##1 1'b1)
```

It is also possible to define sequences that admit only empty matches. For example:

```
1'b1[*0]
```

A sequence that admits no match or that admits only empty matches is called *degenerate*. A sequence that admits at least one nonempty match is called *nondegenerate*. A more precise definition of nondegeneracy is given in Annex E.

The following restrictions apply:

a) Any sequence that is used as a property must be nondegenerate and must not admit any empty match.

b) Any sequence that is used as the antecedent of an overlapping implication (`|->`) must be nondegenerate.

c) Any sequence that is used as the antecedent of a nonoverlapping implication (`|=>`) must admit at least one match. Such a sequence can admit only empty matches.

The reason for these restrictions is that the use of degenerate sequences in the forbidden ways results in counterintuitive property semantics, especially when the property is combined with a **disable iff** clause.

17.12 Multiclock support

Multiclock sequences and properties can be specified using the following syntax.

17.12.1 Multiclocked sequences

Multiclocked sequences are built by concatenating singly clocked subsequences using the single-delay concatenation operator ##1. This operator is nonoverlapping and synchronizes between the clocks of the two sequences. The single delay indicated by ##1 is understood to be from the end point of the first sequence, which occurs at a tick of the first clock, to the nearest strictly subsequent tick of the second clock, where the second sequence begins.

For example, consider

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

A match of this sequence starts with a match of sig0 at **posedge** clk0. Then ##1 moves the time to the nearest strictly subsequent **posedge** clk1, and the match of the sequence ends at that point with a match of sig1. If clk0 and clk1 are not identical, then the clocking event for the sequence changes after ##1. If clk0 and clk1 are identical, then the clocking event does not change after ##1, and the above sequence is equivalent to the singly clocked sequence

```
@(posedge clk0) sig0 ##1 sig1
```

When concatenating differently clocked sequences, the maximal singly clocked subsequences are required to admit only nonempty matches. Thus, if s1, s2 are sequence expressions with no clocking events, then the multiclocked sequence

```
@(posedge clk1) s1 ##1 @(posedge clk2) s2
```

is legal only if neither s1 nor s2 can match the empty word. The clocking event **posedge** clk1 applies throughout the match of s1, while the clocking event **posedge** clk2 applies throughout the match of s2. Because the match of s1 is nonempty, there is an end point of this match at **posedge** clk1. The ##1 synchronizes between this end point and the first occurrence of **posedge** clk2 strictly after it. That occurrence of **posedge** clk2 is the start point of the match of s2.

The restriction that maximal singly clocked subsequences not match the empty word ensures that any multiclocked sequence has well-defined starting and ending clocking events and well-defined clock changes. If clk1 and clk2 are not identical, then the sequence

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1[*0:1]
```

is illegal because of the possibility of an empty match of sig1[*0:1], which would make ambiguous whether the ending clocking event is **posedge** clk0 or **posedge** clk1.

Differently clocked or multiclocked sequence operands cannot be combined with any sequence operators other than ##1. For example, if clk1 and clk2 are not identical, then the following are illegal:

```
@(posedge clk1) s1 ##0 @(posedge clk2) s2

@(posedge clk1) s1 ##2 @(posedge clk2) s2

@(posedge clk1) s1 intersect @(posedge clk2) s2
```

17.12.2 Multiclocked properties

As in the case of singly clocked properties, the result of evaluating a multiclocked property is either true or false. Multiclocked properties can be formed in a number of ways.

Multiclocked sequences are themselves multiclocked properties. For example:

```
@(posedge clk0) sig0 ##1 @(posedge clk1) sig1
```

is a multiclocked property. If a multiclocked sequence is evaluated as a property starting at some point, the evaluation returns true if, and only if, there is a match of the multiclocked sequence beginning at that point.

The boolean property operators (**not**, **and**, **or**) can be used freely to combine singly clocked and multiclocked properties. The meanings of the boolean property operators are the usual ones, just as in the case of singly clocked properties. For example:

```
(@(posedge clk0) sig0) and (@(posedge clk1) sig1)
```

is a multiclocked property, but it is not a multiclocked sequence. This property evaluates to true at a point if, and only if, the two sequences

```
@(posedge clk0) sig0
```

and

```
@(posedge clk1) sig1
```

both have matches beginning at the point.

The nonoverlapping implication operator `|=>` can be used freely to create a multiclocked property from an antecedent sequence and a consequent property that are differently clocked or multiclocked. The meaning of multiclocked nonoverlapping implication is similar to that of singly clocked nonoverlapping implication. For example, if `s0` and `s1` are sequences with no clocking event, then in

```
@(posedge clk0) s0 |=> @(posedge clk1) s1
```

`|=>` synchronizes between **posedge** clk0 and **posedge** clk1. Starting at the point at which the implication is being evaluated, for each match of `s0` clocked by `clk0`, time is advanced from the end point of the match to the nearest strictly future occurrence of **posedge** clk1, and from that point there must exist a match of `s1` clocked by `clk1`.

The nonoverlapping implication operator `|=>` can synchronize between the ending clock event of its antecedent and several leading clock events for subproperties of its consequent. For example, in

```
@(posedge clk0) s0 |=> (@(posedge clk1) s1) and (@(posedge clk2) s2)
```

`|=>` synchronizes between **posedge** clk0 and both **posedge** clk1 and **posedge** clk2.

Because synchronization between distinct clocks always requires strict advance of time, the two property building operators that require special care with multiple clocks are the overlapping implication `|->` and **if**/**if**...**else**.

Because `|->` overlaps the end of its antecedent with the beginning of its consequent, the clock for the end of the antecedent must be the same as the clock for the beginning of the consequent. For example, if `clk0` and `clk1` are not identical and `s0`, `s1`, and `s2` are sequences with no clocking events, then

```
@(posedge clk0) s0 |-> @(posedge clk1) s1 ##1 @(posedge clk2) s2
```

is illegal, but

```
@(posedge clk0) s0 |-> @(posedge clk0) s1 ##1 @(posedge clk2) s2
```

is legal.

The **if**/**if**...**else** operators overlap the test of the boolean condition with the beginning of the **if** clause property and, if present, the **else** clause property. Therefore, whenever using **if** or **if**...**else**, the **if** and **else** clause properties must begin on the same clock as the test of the boolean condition. For example, if `clk0` and `clk1` are not identical and `s0`, `s1`, and `s2` are sequences with no clocking events, then

```
@(posedge clk0) if (b) @(posedge clk0) s1
```

is legal, but

```
@(posedge clk0) if (b) @(posedge clk0) s1 else @(posedge clk1) s2
```

is illegal because the **else** clause property begins on a different clock from the **if** condition.

17.12.3 Clock flow

Throughout this subclause, *c* and *d* denote clocking event expressions and *v*, *w*, *x*, *y*, and *z* denote sequences with no clocking events.

Clock flow allows the scope of a clocking event to extend in a natural way through various parts of multi-clocked sequences and properties and reduces the number of places at which the same clocking event must be specified.

Intuitively, clock flow provides that in a multiclocked sequence or property, the scope of a clocking event flows left to right across linear operators (e.g., repetition, concatenation, negation, implication) and distributes to the operands of branching operators (e.g., conjunction, disjunction, intersection, **if**...**else**) until it is replaced by a new clocking event.

For example:

```
@(c) x |=> @(c) y ##1 @(d) z
```

can be written more simply as

```
@(c) x |=> y ##1 @(d) z
```

because clock *c* is understood to flow across |=>.

Clock flow eliminates the need to write clocking events in positions where the clock is not allowed to change. For example:

```
@(c) x |-> @(c) y ##1 @(d) z
```

can be written as

```
@(c) x |-> y ##1 @(d) z
```

to reinforce the restriction that the clock not change across |->. Similarly,

```
@(c) if (b) @(c) w ##1 @(d) x else @(c) y ##1 @(d) z
```

can be written as

```
@(c) if (b) w ##1 @(d) x else y ##1 @(d) z
```

to reinforce the restriction that the clock not change from the boolean condition *b* to the beginnings of the **if** and **else** clause properties.

Clock flow also makes the adjointness relationships between concatenation and implication clean for multi-clocked properties:

```
@(c) x ##1 y |=> @(d) z
```

is equivalent to

```
@(c)  x  |=> y  |=> @(d)  z
```

and

```
@(c)  x  ##0  y  |=> @(d)  z
```

is equivalent to

```
@(c)  x  |-> y  |=> @(d)  z
```

The scope of a clocking event flows into parenthesized subexpressions and, if the subexpression is a sequence, also flows left to right across the parenthesized subexpression. However, the scope of a clocking event does not flow out of enclosing parentheses.

For example, in

```
@(c)  w  ##1  (x  ##1  @(d)  y)  |=> z
```

w, x, and z are clocked at c, and y is clocked at d. Clock c flows across ##1, across the parenthesized subsequence (x ##1 @(d) y), and across |=>. Clock c also flows into the parenthesized subsequence, but it does not flow through @(d). Clock d does not flow out of its enclosing parentheses.

As another example, in

```
@(c)  v  |=> (w  ##1  @(d)  x) and (y  ##1  z)
```

v, w, y, and z are clocked at c, and x is clocked at d. Clock c flows across |=>, distributes to both operands of the **and** (which is a property conjunction due to the multiple clocking), and flows into each of the parenthesized subexpressions. Within (w ##1 @(d) x), c flows across ##1 but does not flow through @(d). Clock d does not flow out of its enclosing parentheses. Within (y ##1 z), c flows across ##1.

Similarly, the scope of a clocking event flows into an instance of a named property. The scope of a clocking event flows into an instance of a named sequence provided neither method `ended` nor method `matched` is applied to the instance of the sequence. The scope of a clocking event flows left to right across an instance of a sequence, regardless of whether method `ended` or method `matched` is applied. A clocking event in the declaration of a sequence or property does not flow out of an instance of that sequence or property.

The scope of a clocking event does not flow into the reset condition of **disable iff**.

Juxtaposing two clocking events nullifies the first of them; therefore, the two-clocking-event statement

```
@(d)  @(c)  x
```

is equivalent to

```
@(c)  x
```

because the flow of clock d is immediately overridden by clock c.

17.12.4 Examples

The following are examples of multiclock specifications:

```
sequence s1;
   a ##1 b; // unclocked sequence
```

```
    endsequence
    sequence s2;
       c ##1 d; // unclocked sequence
    endsequence
```

a)   Multiclock sequence

```
        sequence mult_s;
           @(posedge clk) a ##1 @(posedge clk1) s1 ##1 @(posedge clk2) s2;
        endsequence
```

b)   Property with a multiclock sequence

```
        property mult_p1;
           @(posedge clk) a ##1 @(posedge clk1) s1 ##1 @(posedge clk2) s2;
        endproperty
```

c)   Property with a named multiclock sequence

```
        property mult_p2;
           mult_s;
        endproperty
```

d)   Property with multiclock implication

```
        property mult_p3;
           @(posedge clk) a ##1 @(posedge clk1) s1 |=> @(posedge clk2) s2;
        endproperty
```

e)   Property with implication, where antecedent and consequent are named multiclocked sequences

```
        property mult_p6;
           mult_s |=> mult_s;
        endproperty
```

f)   Property using clock flow and overlapped implication:

```
        property mult_p7;
           @(posedge clk) a ##1 b |-> c ##1 @(posedge clk1) d;
        endproperty
```

      Here, a, b, and c are clocked at **posedge** clk.

g)   Property using clock flow and **if**...**else**:

```
        property mult_p8;
           @(posedge clk) a ##1 b |->
           if (c)
              (1 |=> @(posedge clk1) d)
           else
              e ##1 @(posedge clk2) f ;
        endproperty
```

      Here, a, b, c, e, and constant 1 are clocked at **posedge** clk.

17.12.5 Detecting and using end point of a sequence in multiclock context

Method `ended` can be applied to detect the end point of a multiclocked sequence. Method `ended` can also be applied to detect the end point of a sequence from within a multiclocked sequence. In both cases, the ending clock of the sequence instance to which `ended` is applied must be the same as the clock in the context where the application of method `ended` appears.

To detect the end point of a sequence when the clock of the source sequence is different from the destination sequence, method `matched` on the source sequence is used. The end point of a sequence is reached whenever there is a match on its expression.

The syntax of the `matched` method is as follows:

```
sequence_instance.matched
```

`matched` is a method on a sequence that returns true or false. Unlike `ended`, `matched` uses synchronization between the two clocks, by storing the result of the source sequence match until the arrival of the first destination clock tick after the match. The result of `matched` does not depend upon the starting point of the source sequence.

Like `ended`, `matched` can be used on sequences that have formal arguments.

An example is shown below:

```
sequence e1(a,b,c);
    @(posedge clk) $rose(a) ##1 b ##1 c ;
endsequence
sequence e2;
    @(posedge sysclk) reset ##1 inst ##1 e1(ready,proc1,proc2).matched [->1]
        ##1 branch_back;
endsequence
```

In this example, source sequence `e1` is evaluated at clock `clk`, while the destination sequence `e2` is evaluated at clock `sysclk`. In `e2`, the end point of the instance `e1(ready,proc1,proc2)` is tested to occur sometime after the occurrence of `inst`. Notice that method `matched` only tests for the end point of `e1(ready,proc1,proc2)` and has no bearing on the starting point of `e1(ready,proc1,proc2)`.

Local variables can be passed into an instance of a named sequence to which `matched` is applied. The same restrictions apply as in the case of `ended`. Values of local variables sampled in an instance of a named sequence to which `matched` is applied will flow out under the same conditions as for `ended`. See 17.8.

As with `ended`, a sequence instance to which `matched` is applied can have multiple matches in a single cycle of the destination sequence clock. The multiple matches are treated semantically the same way as matching both disjuncts of an **or**. In other words, the thread evaluating the destination sequence will fork to account for such distinct local variable valuations.

17.12.6 Sequence methods

There are three methods available to identify the end point of a sequence: `ended`, `triggered`, and `matched`. These methods are invoked using the following syntax:

```
sequence_instance.sequence_method
```

The results of these operations are true or false and do not depend upon the starting point of the match of their operand sequence. These methods can be invoked on sequences with formal arguments.

The value of method `ended` evaluates to true if the given sequence has reached its end point at that particular point in time and false otherwise. The ended status of the sequence is set in the Observe region and persists through the Observe region. This method shall only be used to detect the end point of a sequence used in another sequence. It shall be considered an error if this method is used in **disable iff** boolean expression for properties. There shall be no circular dependencies between sequences induced by the use of `ended`.

The value of method `triggered` evaluates to true if the given sequence has reached its end point at that particular point in time and false otherwise. The triggered status of the sequence is set in the Observe region and persists through the remainder of the time step. This method shall only be used in **wait** statements or boolean expressions (see 10.11) outside of sequence context or in the **disable iff** boolean expression for properties. It shall be considered an error to invoke this method on sequences that treat their formal arguments as local variables. A sequence treats its formal argument as a local variable if the formal argument is used as an lvalue in operator_assignment or inc_or_dec_expression in sequence_match_item.

Unlike `ended` and `triggered`, `matched` provides synchronization between two clocks by storing the result of the source sequence until the arrival of the first clock tick of the destination sequence after the match. The matched status of the sequence is set in the Observe region and persists until the Observe region following the arrival of the first clock tick of the destination sequence after the match. This method is used to detect the end point of a sequence used in a multiclocked sequence. Like `ended`, `matched` can only be used in sequence expressions.

It shall be considered an error to use sequence methods in sampled value functions (see 17.7.3) because the values of sequence methods are not available in the Preponed region.

An example of using the above methods on a sequence is shown below:

```
sequence e1;
   @(posedge sysclk) $rose(a) ##1 b ##1 c;
endsequence

sequence e2;
   @(posedge sysclk) reset ##1 inst ##1 e1.ended ##1 branch_back;
endsequence

sequence e3;
   @(posedge clk) reset1 ##1 e1.matched ##1 branch_back1;
endsequence


program check;
   initial begin
      wait (e1.triggered || e2.triggered);
      if (e1.triggered)
         $display("e1 passed");
      if (e2.triggered)
         $display("e2 passed");
      L2: ...
   end
endprogram
```

In the example above, sequence `e2` tests for the end point of sequence `e1` using method `ended` because both sequences use the same clock. The sequence `e3` tests for the end point of sequence `e1` using method `matched` because `e1` and `e3` use different clocks. The **initial** block in the program waits for the end point of either `e1` or `e2`. When either `e1` or `e2` evaluates to true, the wait statement unblocks the initial process. The process then displays the sequence that caused it to unblock, and then continues to execute at the statement labeled `L2`.

More details about sequence methods can be found in <u>10.11</u>, <u>17.7.10</u>, and <u>17.12.5</u>.

17.13 Concurrent assertions

A property on its own is never evaluated for checking an expression. It must be used within a verification statement for this to occur. A verification statement states the verification function to be performed on the property. The statement can be one of the following:

— **assert** to specify the property as a checker to ensure that the property holds for the design
— **assume** to specify the property as an assumption for the environment
— **cover** to monitor the property evaluation for coverage

A concurrent assertion statement can be specified in any of the following:

— An **always** block or **initial** block as a statement, wherever these blocks can appear
— A module
— An interface
— A program

---

procedural_assertion_statement ::=   *// from <u>A.6.10</u>*
        concurrent_assertion_statement
      | immediate_assert_statement

concurrent_assertion_item ::= [block_identifier **:**] concurrent_assertion_statement   *// from <u>A.2.10</u>*

concurrent_assertion_statement ::=
        assert_property_statement
      | assume_property_statement
      | cover_property_statement

assert_property_statement::=
        assert property (property_spec **)** action_block

assume_property_statement::=
        assume property (property_spec **) ;**

cover_property_statement::=
        cover property (property_spec **)** statement_or_null

---

Syntax 17-16—Concurrent assert construct syntax (excerpt from <u>Annex A</u>)

The **assert**, **assume**, or **cover** statements can be referenced by their optional name. A hierarchical name can be used consistent with the SystemVerilog naming conventions. When a name is not provided, a tool shall assign a name to the statement for the purpose of reporting. Assertion control system tasks are described in <u>22.8</u>.

17.13.1 Assert statement

The **assert** statement is used to enforce a **property** as a checker. When the property for the **assert** statement is evaluated to be true, the pass statements of the action block are executed. Otherwise, the fail statements of the *action_block* are executed. For example:

```
property abc(a,b,c);
    disable iff (a==2) @(posedge clk) not (b ##1 c);
endproperty
env_prop: assert property (abc(rst,in1,in2))
            $display("env_prop passed."); else $display("env_prop failed.");
```

When no action is needed, a null statement (i.e., `;`) is specified. If no statement is specified for **else**, then `$error` is used as the statement when the assertion fails.

The *action_block* shall not include any concurrent **assert**, **assume**, or **cover** statement. The *action_block*, however, can contain immediate assertion statements.

The pass and fail statements of an assert statement are executed in the Reactive region. The regions of execution are explained in the scheduling semantics in Clause 9.

17.13.2 Assume statement

The purpose of the **assume** statement is to allow properties to be considered as assumptions for formal analysis as well as for dynamic simulation tools. When a property is assumed, the tools constrain the environment so that the property holds.

For formal analysis, there is no obligation to verify that the assumed properties hold. An assumed property can be considered as a hypothesis to prove the asserted properties.

For simulation, the environment must be constrained so that the properties that are assumed shall hold. Like an assert property, an assumed property must be checked and reported if it fails to hold. There is no requirement on the tools to report successes of the assumed properties.

Additionally, for random simulation, biasing on the inputs provides a way to make random choices. An expression can be associated with biasing as shown below.

```
expression dist { dist_list } ;  // from A.1.9
```

Distribution sets and the **dist** operator are explained in 13.4.4.

The biasing feature is only useful when properties are considered as assumptions to drive random simulation. When a property with biasing is used in an assertion or coverage, the **dist** operator is equivalent to inside operator, and the weight specification is ignored. For example:

```
a1:assume property ( @(posedge clk) req dist {0:=40, 1:=60} ) ;
property proto ;
    @(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty
```

This is equivalent to the following:

```
a1_assertion:assert property ( @(posedge clk) req inside {0, 1} ) ;
property proto_assertion ;
    @(posedge clk) req |-> req[*1:$] ##0 ack;
endproperty
```

In the above example, signal `req` is specified with distribution in assumption `a1` and is converted to an equivalent assertion `a1_assertion`.

It should be noted that the properties that are assumed must hold in the same way with or without biasing. When using an assume statement for random simulation, the biasing simply provides a means to select values of free variables, according to the specified weights, when there is a choice of selection at a particular time.

Consider an example specifying a simple synchronous request and acknowledge protocol, where variable `req` can be raised at any time and must stay asserted until `ack` is asserted. In the next clock cycle, both `req` and `ack` must be deasserted.

Properties governing `req` are as follows:

```
property pr1;
    @(posedge clk) !reset_n |-> !req; //when reset_n is asserted (0),keep req
0
endproperty
property pr2;
    @(posedge clk) ack |=> !req; // one cycle after ack, req must be
deasserted
endproperty
property pr3;
    @(posedge clk) req |-> req[*1:$] ##0 ack; // hold req asserted until
                                        // and including ack asserted

endproperty
```

Properties governing `ack` are as follows:

```
property pa1;
    @(posedge clk) !reset_n || !req |-> !ack;
endproperty
property pa2;
    @(posedge clk) ack |=> !ack;
endproperty
```

When verifying the behavior of a protocol controller that has to respond to requests on `req`, assertions `assert_ack1` and `assert_ack2` should be proven while assuming that statements `a1`, `assume_req1`, `assume_req2`, and `assume_req3` hold at all times.

```
a1:assume property @(posedge clk) req dist {0:=40, 1:=60} ;
assume_req1:assume property (pr1);
assume_req2:assume property (pr2);
assume_req3:assume property (pr3);

assert_ack1:assert property (pa1)
    else $display("\n ack asserted while req is still deasserted");
assert_ack2:assert property (pa2)
    else $display("\n ack is extended over more than one cycle");
```

The **assume** statement does not provide an action block, as the actions for an assumption serve no purpose.

17.13.3 Cover statement

To monitor sequences and other behavioral aspects of the design for coverage, the same syntax is used with the **cover** statement. The tools can gather information about the evaluation and report the results at the end of simulation. When the property for the **cover** statement is successful, the pass statements can specify a coverage function, such as monitoring all paths for a sequence. The pass statement shall not include any concurrent **assert**, **assume**, or **cover** statement.

Coverage results are divided into two categories: coverage for properties and coverage for sequences.

For sequence coverage, the statement appears as follows:

```
cover property ( sequence_expr ) statement_or_null
```

The results of coverage statement for a property shall contain the following:

— Number of times attempted
— Number of times succeeded
— Number of times failed
— Number of times succeeded because of vacuity

In addition, `statement_or_null` is executed every time a property succeeds.

Vacuity rules are applied only when implication operator is used. A property succeeds nonvacuously only if the consequent of the implication contributes to the success.

Results of coverage for a sequence shall include the following:

— Number of times attempted
— Number of times matched (each attempt can generate multiple matches)

In addition, `statement_or_null` gets executed for every match. If there are multiple matches at the same time, the statement gets executed multiple times, one for each match.

17.13.4 Using concurrent assertion statements outside of procedural code

A concurrent assertion statement can be used outside of a procedural context. It can be used within a module, an interface, or a program. A concurrent assertion statement is an **assert**, an **assume**, or a **cover** statement. Such a concurrent assertion statement uses the **always** semantics.

The following two forms are equivalent:

```
assert property ( property_spec ) action_block

always assert property ( property_spec ) action_block ;
```

Similarly, the following two forms are equivalent:

```
cover property ( property_spec ) statement_or_null

always cover property ( property_spec ) statement_or_null
```

For example:

```
module top(input bit clk);
   logic a,b,c;
   property rule3;
      @(posedge clk) a |-> b ##1 c;
   endproperty
   a1: assert property (rule3);
   ...
endmodule
```

rule3 is a property declared in module `top`. The assert statement `a1` starts checking the property from the beginning to the end of simulation. The property is always checked. Similarly,

```
module top(input bit clk);
   logic a,b,c;
```

```
      sequence seq3;
         @(posedge clk) b ##1 c;
      endsequence
      c1: cover property (seq3);
      ...
   endmodule
```

The cover statement `c1` starts coverage of the sequence `seq3` from beginning to the end of simulation. The sequence is always monitored for coverage.

17.13.5 Embedding concurrent assertions in procedural code

A concurrent assertion statement can also be embedded in a procedural block. For example:

```
   property rule;
      a ##1 b ##1 c;
   endproperty

   always @(posedge clk) begin
      <statements>
      assert property (rule);
   end
```

If the statement appears in an **always** block, the property is always monitored. If the statement appears in an **initial** block, then the monitoring is performed only on the first clock tick.

Two inferences are made from the procedural context: the clock from the event control of an **always** block and the enabling conditions.

A clock is inferred if the statement is placed in an **always** or **initial** block with an event control abiding by the following rules:

— The clock to be inferred must be placed as the first term of the event control as an edge specifier (**posedge** *expression* or **negedge** *expression*).

— The variables in *expression* must not be used anywhere in the **always** or **initial** block.

For example:

```
   property r1;
      q != d;
   endproperty
   always @(posedge mclk) begin
      q <= d1;
      r1_p: assert property (r1);
   end
```

The above property can be checked by writing statement `r1_p` outside the **always** block and declaring the property with the clock as follows:

```
   property r1;
      @(posedge mclk)q != d;
   endproperty
   always @(posedge mclk) begin
      q <= d1;
   end
   r1_p: assert property (r1);
```

If the clock is explicitly specified with a property, then it must be identical to the inferred clock, as shown below:

```
property r2;
    @(posedge mclk)(q != d);
endproperty
always @(posedge mclk) begin
    q <= d1;
    r2_p: assert property (r2);
end
```

In the above example, (posedge mclk) is the clock for property r2.

Another inference made from the context is the enabling condition for a property. Such derivation takes place when a property is placed in an if...else block or a case block. The enabling condition assumed from the context is used as the antecedent of the property.

```
property r3;
    @(posedge mclk)(q != d);
endproperty
always @(posedge mclk) begin
    if (a) begin
        q <= d1;
        r3_p: assert property (r3);
    end
end
```

The above example is equivalent to the following:

```
property r3;
    @(posedge mclk)a |-> (q != d);
endproperty
r3_p: assert property (r3);
always @(posedge mclk) begin
    if (a) begin
        q <= d1;
    end
end
```

Similarly, the enabling condition is also inferred from case statements.

```
property r4;
    @(posedge mclk)(q != d);
endproperty
always @(posedge mclk) begin
    case (a)
        1: begin q <= d1;
               r4_p: assert property (r4);
            end
        default: q1 <= d1;
    endcase
end
```

The above example is equivalent to the following:

```
property r4;
    @(posedge mclk)(a==1) |-> (q != d);
```

```
      endproperty
   r4_p: assert property (r4);
   always @(posedge mclk) begin
      case (a)
         1: begin q <= d1;
            end
         default: q1 <= d1;
      endcase
   end
```

The enabling condition is inferred from procedural code inside an **always** or **initial** block, with the following restrictions:

a)   There must not be a preceding statement with a timing control.

b)   A preceding statement shall not invoke a task call that contains a timing control on any statement.

c)   The concurrent assertion statement shall not be placed in a looping statement, immediately, or in any nested scope of the looping statement.

17.14 Clock resolution

There are a number of ways to specify a clock for a property:

—   Sequence instance with a clock, for example:

```
      sequence s2; @(posedge clk) a ##2 b; endsequence
      property p2; not s2; endproperty
      assert property (p2);
```

—   Property, for example:

```
      property p3; @(posedge clk) not (a ##2 b); endproperty
      assert property (p3);
```

—   Contextually inferred clock from a procedural block, for example:

```
      always @(posedge clk) assert property (not (a ##2 b));
```

—   A **clocking** block, for example:

```
      clocking master_clk @(posedge clk);
         property p3; not (a ##2 b); endproperty
      endclocking
      assert property (master_clk.p3);
```

—   Default clock, for example:

```
      default clocking master_clk ;  // master clock as defined above
      property p4; (a ##2 b); endproperty
      assert property (p4);
```

In general, a clocking event applies throughout its scope except where superseded by an inner clocking event, as with clock flow in multiclocked sequences and properties. The following rules apply:

a)   In a module, interface, or program with a default clocking event, a concurrent assertion statement that has no otherwise specified leading clocking event is treated as though the default clocking event

had been written explicitly as the leading clocking event. The default clocking event does not apply to a sequence or property declaration except in the case that the declaration appears in a **clocking** block whose clocking event is the default.

b) The following rules apply within a **clocking** block:

1) No explicit clocking event is allowed in any property or sequence declaration within the **clocking** block. All sequence and property declarations within the **clocking** block are treated as though the clocking event of the **clocking** block had been written explicitly as the leading clocking event.

2) Multiclocked sequences and properties are not allowed within the **clocking** block.

3) If a named sequence or property that is declared outside the **clocking** block is instantiated within the **clocking** block, the instance must be singly clocked and its clocking event must be identical to that of the **clocking** block.

c) A contextually inferred clocking event from a procedural block supersedes a default clocking event. The contextually inferred clocking event is treated as though it had been written as the leading clocking event of any concurrent assertion statement to which the inferred clock applies. The maximal property of such a concurrent assertion statement must be singly clocked, and the clocking event, if specified otherwise, must be identical to the contextually inferred clocking event.

d) An explicitly specified leading clocking event in a concurrent assertion statement supersedes a default clocking event.

e) A multiclocked sequence or property can inherit the default clocking event as its leading clocking event. If a multiclocked property is the maximal property of a concurrent assertion statement, then the property must have a unique semantic leading clock (see 17.14.1).

f) If a concurrent assertion statement has no explicit leading clocking event, there is no default clocking event, and no contextually inferred clocking event applies to the assertion statement, then the maximal property of the assertion statement must be an instance of a sequence or property for which a unique leading clocking event is determined.

Below are two example modules illustrating the application of these rules with some legal and some illegal declarations, as indicated by the comments.

```
module examples_with_default (input logic a, b, c, clk);

  property q1;
     $rose(a) |-> ##[1:5] b;
  endproperty

  property q2;
     @(posedge clk) q1;
  endproperty

  default clocking posedge_clk @(posedge clk);
     property q3;
        $fell(c) |=> q1;
        // legal: q1 has no clocking event
     endproperty

     property q4;
        $fell(c) |=> q2;
        // legal: q2 has clocking event identical to that of
        // the clocking block
     endproperty

     sequence s1;
        @(posedge clk) b[*3];
```

```
                  // illegal: explicit clocking event in clocking block
          endsequence
      endclocking

      property q5;
          @(negedge clk) b[*3] |=> !b;
      endproperty

      always @(negedge clk)
      begin
         a1: assert property ($fell(c) |=> q1);
            // legal: contextually inferred leading clocking event,
            // @(negedge clk)
         a2: assert property (posedge_clk.q4);
            // illegal: clocking event of posedge_clk.q4 not identical
            // to contextually inferred leading clocking event
         a3: assert property ($fell(c) |=> q2);
            // illegal: multiclocked property with contextually
            // inferred leading clocking event
         a4: assert property (q5);
            // legal: contextually inferred leading clocking event,
            // @(negedge clk)
      end

      property q6;
          q1 and q5;
      endproperty

      a5: assert property (q6);
         // illegal: default leading clocking event, @(posedge clk),
         // but semantic leading clock is not unique
      a6: assert property ($fell(c) |=> q6);
         // legal: default leading clocking event, @(posedge clk),
         // is the unique semantic leading clock

      sequence s2;
          $rose(a) ##[1:5] b;
      endsequence

      c1: cover property (s2);
         // legal: default leading clocking event, @(posedge clk)
      c2: cover property (@(negedge clk) s2);
         // legal: explicit leading clocking event, @(negedge clk)

   endmodule


   module examples_without_default (input logic a, b, c, clk);

      property q1;
          $rose(a) |-> ##[1:5] b;
      endproperty

      property q5;
          @(negedge clk) b[*3] |=> !b;
      endproperty

      property q6;
          q1 and q5;
```

```
    endproperty

a5: assert property (q6);
    // illegal: no leading clocking event
a6: assert property ($fell(c) |=> q6);
    // illegal: no leading clocking event

sequence s2;
    $rose(a) ##[1:5] b;
endsequence

c1: cover property (s2);
    // illegal: no leading clocking event
c2: cover property (@(negedge clk) s2);
    // legal: explicit leading clocking event, @(negedge clk)

sequence s3;
    @(negedge clk) s2;
endsequence

c3: cover property (s3);
    // legal: leading clocking event, @(negedge clk),
    // determined from declaration of s3
c4: cover property (s3 ##1 b);
    // illegal: no default, inferred, or explicit leading
    // clocking event and maximal property is not an instance

endmodule
```

17.14.1 Clock resolution in multiclocked properties

Throughout this subclause, s, s_1, and s_2 denote sequences without clocking events; p, p_1, and p_2 denote properties without clocking events; m, m_1, and m_2 denote multiclocked sequences, q, q_1, and q_2 denote multiclocked properties; and c, c_1, and c_2 denote nonidentical clocking event expressions.

Due to clock flow, juxtaposition of two clocks nullifies the first. This and the nesting of clocking events within other property building operators mean that there are subtleties in the general interpretation of the restrictions about where the clock can change in multiclocked properties. For example:

```
@(c) s |-> @(c) (p and @(c₁) p₁)
```

appears legal because the antecedent is clocked by c and the consequent begins syntactically with the clocking event `@(c)`. However, the consequent sequence is equivalent to

```
(@(c) p) and (@(c₁) p₁)
```

and `|->` cannot synchronize between clock c from the antecedent and clock c_1 from the second conjunct of the consequent. Similarly,

```
@(c) s |-> @(c₁) (@(c) p)
```

appears illegal due to the apparent clock change from c to c_1 across `|->`. However, it is legal, although arguably misleading in style, because the consequent property is equivalent to `@(c) p`.

This subclause gives a more precise treatment of the restrictions on multiclocked use of `|->` and **if**/**if**...**else** than the intuitive discussion in 17.12. The present treatment depends on the notion of the set of semantic leading clocks for a multiclocked sequence or property.

Some sequences and properties have no explicit leading clock event. Their initial clocking event is inherited from an outer clocking event according to the flow of clocking event scope. In this case, the semantic leading clock is said to be *inherited*. For example, in the property

```
@(c) s |=> p and @(c₁) p₁
```

the semantic leading clock of the subproperty p is inherited because the initial clock of p is the clock that flows across |=>.

A multiclocked sequence has a unique semantic leading clock, defined inductively as follows:

— The semantic leading clock of s is inherited.

— The semantic leading clock of @(c) s is c.

— If inherited is the semantic leading clock of m, then the semantic leading clock of @(c) m is c. Otherwise, the semantic leading clock of @(c) m is equal to the semantic leading clock of m.

— The semantic leading clock of (m) is equal to the semantic leading clock of m.

— The semantic leading clock of m_1 ##1 m_2 is equal to the semantic leading clock of m_1.

The set of semantic leading clocks of a multiclocked property is defined inductively as follows:

— The set of semantic leading clocks of m is $\{c\}$, where c is the unique semantic leading clock of m.

— The set of semantic leading clocks of p is {*inherited*}.

— If inherited is an element of the set of semantic leading clocks of q, then the set of semantic leading clocks of @(c) q is obtained from the set of semantic leading clocks of q by replacing *inherited* by c. Otherwise, the set of semantic leading clocks of @(c) q is equal to the set of semantic leading clocks of q.

— The set of semantic leading clocks of (q) is equal to the set of semantic leading clocks of q.

— The set of semantic leading clocks of **not** q is equal to the set of semantic leading clocks of q.

— The set of semantic leading clocks of q_1 and q_2 is the union of the set of semantic leading clocks of q_1 with the set of semantic leading clocks of q_2.

— The set of semantic leading clocks of q_1 or q_2 is the union of the set of semantic leading clocks of q_1 with the set of semantic leading clocks of q_2.

— The set of semantic leading clocks of m |-> p is equal to the set of semantic leading clocks of m.

— The set of semantic leading clocks of m |=> p is equal to the set of semantic leading clocks of m.

— The set of semantic leading clocks of **if** (b) q is {*inherited*}.

— The set of semantic leading clocks of **if** (b) q_1 else q_2 is {*inherited*}.

— The set of semantic leading clocks of a property instance is equal to the set of semantic leading clocks of the multiclocked property obtained from the body of its declaration by substituting in actual arguments.

For example, the multiclocked sequence

```
@(c₁) s₁ ##1 @(c₂) s₂
```

has c_1 as its unique semantic leading clock, while the multiclocked property

```
not (p₁ and (@(c₂) p₂))
```

has {*inherited*, c_2} as its set of semantic leading clocks.

In the presence of an incoming outer clock, the inherited semantic leading clock is always understood to refer to the incoming outer clock. Therefore, the clocking of a property q in the presence of incoming outer clock c is equivalent to the clocking of the property `@(c) q`.

The rules for using multiclocked overlapping implication and **if**/**if**...**else** in the presence of an incoming outer clock can now be stated more precisely.

a) Multiclocked overlapping implication.

   Let c be the incoming outer clock. Then the clocking of m `|->` q is equivalent to the clocking of `@(c)` m `|->` q

   In the presence of the incoming outer clock, m has a well-defined ending clock, and there is a well-defined clock that flows across `|->`. The multiclocked overlapped implication m `|->` q is legal for incoming clock c if, and only if, the following two conditions are met:

   1) Every explicit semantic leading clock of q is identical to the ending clock of m.

   2) If *inherited* is a semantic leading clock of q, then the ending clock of m is equal to the clock that flows across `|->`.

   For example:

```
@(c) s |-> p₁ or @(c₂) p₂
```

   is not legal because the ending clock of the antecedent is c, while the consequent has c_2 as an explicit semantic leading clock.

   Also,

```
@(c) s ##1 (@(c₁) s₁) |-> p
```

   is not legal because the set of semantic leading clocks of p is {*inherited*}, the ending clock of the antecedent is c_1, and the clock that flows across `|->` and is inherited by p is c.

   On the other hand,

```
@(c) s |-> p₁ or @(c) p₂
```
   and

```
@(c) s ##1 @(c₁) s₁ |-> p₁ or @(c₁) p₂
```

   are both legal.

b) Multiclocked **if**/**if**...**else**

   Let c be the incoming outer clock. Then the clocking of **if** (b) q_1 [**else** q_2] is equivalent to the clocking of

```
@(c) if (b) q₁ [ else q₂ ]
```

   The boolean condition b is clocked by c; therefore, the multiclocked **if**/**if**...**else if** (b) q_1 [**else** q_2] is legal for incoming clock c if, and only if, the following condition is met:

   — Every explicit semantic leading clock of q_1 [**or** q_2] is identical to c.

   For example:

```
@(c) if (b) p₁ else @(c) p₂
```

   is legal, but

$$@(c) \text{ if } (b) \ @(c) \ (p_1 \text{ and } @(c_2) \ p_2)$$

is not.

17.15 Binding properties to scopes or instances

To facilitate verification separate from design, it is possible to specify properties and bind them to specific modules or instances. The following are some goals of providing this feature:

— It allows verification engineers to verify with minimum changes to the design code and files.
— It allows a convenient mechanism to attach verification Internet Protocol (IP) to a module or an instance.
— No semantic changes to the assertions are introduced due to this feature. It is equivalent to writing properties external to a module, using hierarchical path names.

With this feature, a user can bind a module, interface, or program instance to a module or a module instance.

The syntax of the **bind** construct is as follows:

```
bind_directive ::=                                                          // from A.1.4
      bind bind_target_scope [: bind_target_instance_list] bind_instantiation ;
    | bind bind_target_instance bind_instantiation ;
bind_target_scope ::=
      module_identifier
    | interface_identifier
bind_target_instance ::=
      hierarchical_identifier constant_bit_select
bind_target_instance_list ::=
      bind_target_instance { , bind_target_instance }
bind_instantiation ::=
      program_instantiation
    | module_instantiation
    | interface_instantiation
```

Syntax 17-17—Bind construct syntax (excerpt from Annex A)

The **bind** directive can be specified in any of the following:

— A module
— An interface
— A compilation-unit scope

There are two forms of bind syntax. In the first form, *bind_target_scope* specifies a target scope into which the *bind_instantiation* should be inserted. Possible target scopes include module, program, and interface declarations. In the absence of a *bind_target_instance_list*, the *bind_instantiation* is inserted into all instances of the specified target scope, designwide. If a *bind_target_instance_list* is present, the *bind_instantiation* is only inserted into the specified instances of the target scope. The *bind_instantiation* is effectively a complete program, module, or interface instantiation statement.

The second form of bind syntax can be used to specify a single instance into which the *bind_instantiation* should be inserted. If the second form of bind syntax is used and the *bind_target_instance* identifier resolves to both an instance name and a module name, binding shall only occur to the specified instance.

Example of binding a program instance to a module:

```
bind cpu fpu_props fpu_rules_1(a,b,c);
```

where

— `cpu` is the name of the target module.
— `fpu_props` is the name of the program to be instantiated.
— `fpu_rules_1` is the program instance name to be created in the target scope.
— An instance named `fpu_rules_1` is instantiated in every instance of module `cpu`.
— The first three ports of program `fpu_props` get bound to objects a, b, and c in module `cpu` (these objects are viewed from module `cpu`'s point of view, and they are completely distinct from any objects named a, b, and c that are visible in the scope that contains the **bind** directive).

Example of binding a program instance to a specific instance of a module:

```
bind cpu: cpu1 fpu_props fpu_rules_1(a, b, c);
```

In the example above, the `fpu_rules_1` instance is bound into the `cpu1` instance of module `cpu`.

Example of binding a program instance to multiple instances of a module:

```
bind cpu: cpu1, cpu2, cpu3 fpu_props fpu_rules_1(a, b, c);
```

In the example above, the `fpu_rules_1` instance is bound into instances `cpu1`, `cpu2`, and `cpu3` of module `cpu`.

By binding a program to a module or an instance, the program becomes part of the bound object. The names of assertion-related declarations can be referenced using the SystemVerilog hierarchical naming conventions.

Binding of a module instance or an interface instance works the same way as described for programs above.

```
interface range (input clk,enable, input int minval,expr);
   property crange_en;
      @(posedge clk) enable |-> (minval <= expr);
   endproperty
   range_chk: assert property (crange_en);
endinterface

bind cr_unit range r1(c_clk,c_en,v_low,(in1&&in2));
```

In this example, interface `range` is instantiated in the module `cr_unit`. Effectively, every instance of module `cr_unit` shall contain the interface instance `r1`.

The *bind_instantiation* portion of the **bind** statement allows the complete range of SystemVerilog instantiation syntax. In other words, both parameter and port associations may appear in the *bind_instantiation*. All actual ports and parameters in the *bind_instantiation* refer to objects from the viewpoint of the *bind_target_instance*.

When an instance is bound into a target scope, the effect will be as if the instance was present at the very end of the target scope. In other words, all declarations present in the target scope are visible to the bound instance.

If multiple **bind** statements are present in a given scope, the order of those statements is not important. An implementation is free to elaborate **bind** statements in any order it chooses.

The following is an example of a module containing a **bind** statement with complex instantiation syntax. All identifiers in the **bind** instantiation are referenced from the bind target's point of view in the overall design hierarchy.

```
bind targetmod
mycheck #(.param1(const4), .param2(8'h44))
i_mycheck(.*, .p1(f1({v1, 1'b0, b1.c}, v2 & v3)), .p2(top.v4));
```

If any controlling configuration library mapping is in effect at the time a **bind** statement is encountered, the mapping associated with the **bind** statement shall influence the elaboration of the *bind_instantiation* statement. In all cases, library mapping associated with the *bind_target_instance* shall be ignored during elaboration of the *bind_instantiation*.

It shall be an error to use noninstance-based binding if the design contains more than one variation of the target module, program, or interface. This can occur in the presence of configuration library mapping or nonstandard functionality such as provided by the `uselib directive. In such cases, users must use instance-based binding syntax to disambiguate between the multiple variations of the target.

Any **defparam** statement located at a lower level of the *bind_instantiation*'s hierarchy must not extend influence outside the scope of that local hierarchy. This is similar to the rules for use of **defparam** inside the scope of generated hierarchy.

Hierarchical references to a *bind_instantiation*'s parameters may not be used outside the instantiation in any context that requires a constant expression. Examples of such contexts include type descriptions and generate conditions.

It is legal for more than one **bind** statement to bind a *bind_instantiation* into the same target scope. However, it shall be an error for a *bind_instantiation* to introduce an instance name that clashes with another name in the module name space of the target scope (see 19.13). This applies to both pre-existing names as well as instance names introduced by other **bind** statements. The latter situation will occur if the design contains more than one instance of a module containing a **bind** statement.

It shall be an error for a **bind** statement to bind a *bind_instantiation* underneath the scope of another *bind_instantiation*.

17.16 Expect statement

The **expect** statement is a procedural blocking statement that allows waiting on a property evaluation. The syntax of the **expect** statement accepts a named property or a property declaration and is given below.

expect_property_statement ::=	*// from A.2.10*
expect (property_spec **)** action_block	

Syntax 17-18—Expect statement syntax (excerpt from Annex A)

The **expect** statement accepts the same syntax used to assert a property. An **expect** statement causes the executing process to block until the given property succeeds or fails. The statement following the **expect** is scheduled to execute after processing the Observe region in which the property completes its evaluation. When the property succeeds or fails, the process unblocks, and the property stops being evaluated (i.e., no property evaluation is started until that **expect** statement is executed again).

When executed, the **expect** statement starts a single thread of evaluation for the given property on the subsequent clocking event, that is, the first evaluation shall take place on the next clocking event. If the property fails at its clocking event, the optional **else** clause of the action block is executed. If the property succeeds, the optional pass statement of the action block is executed.

```
program tst;
    initial begin
        # 200ms;
        expect( @(posedge clk) a ##1 b ##1 c ) else $error( "expect failed" );
        ABC: ...
    end
endprogram
```

In the above example, the **expect** statement specifies a property that consists of the sequence a ##1 b ##1 c. The **expect** statement (second statement in the **initial** block of program tst) blocks until the sequence a ##1 b ##1 c is matched or is determined not to match. The property evaluation starts on the clocking event (**posedge** clk) following the 200 ms delay. If the sequence is matched, the process is unblocked and continues to execute on the statement labeled ABC. If the sequence fails to match, then the **else** clause is executed, which in this case generates a run-time error. For the expect above to succeed, the sequence a ##1 b ##1 c must match starting on the clocking event (**posedge** clk) immediately after time 200ms. The sequence will not match if a, b, or c is evaluated to be false at the first, second, or third clocking event, respectively.

The **expect** statement can be incorporated in any procedural code, including tasks or class methods. Because it is a blocking statement, the property can refer to automatic variables as well as static variables. For example, the task below waits between 1 and 10 clock ticks for the variable data to equal a particular value, which is specified by the automatic argument value. The second argument, success, is used to return the result of the **expect** statement: 1 for success and 0 for failure.

```
integer data;
...
task automatic wait_for( integer value, output bit success );
expect( @(posedge clk) ##[1:10] data == value ) success = 1;
    else success = 0;
endtask

initial begin
    bit ok;
    wait_for( 23, ok );  // wait for the value 23
    ...
end
```

17.17 Clocking blocks and concurrent assertions

If a variable used in a concurrent assertion is a **clocking** block variable, it will be sampled only in the **clocking** block.

Examples:

```
module A;
    logic a, clk;

    clocking cb_with_input @(posedge clk);
        input a;
        property p1;
            a;
```

```
        endproperty
    endclocking

    clocking cb_without_input @(posedge clk);
        property p1;
            a;
        endproperty
    endclocking

    property p1;
        @(posedge clk) a;
    endproperty

    property p2;
        @(posedge clk) cb_with_input.a;
    endproperty

    a1: assert property (p1);
    a2: assert property (cb_with_input.p1);
    a3: assert property (p2);
    a4: assert property (cb_without_input.p1);

endmodule
```

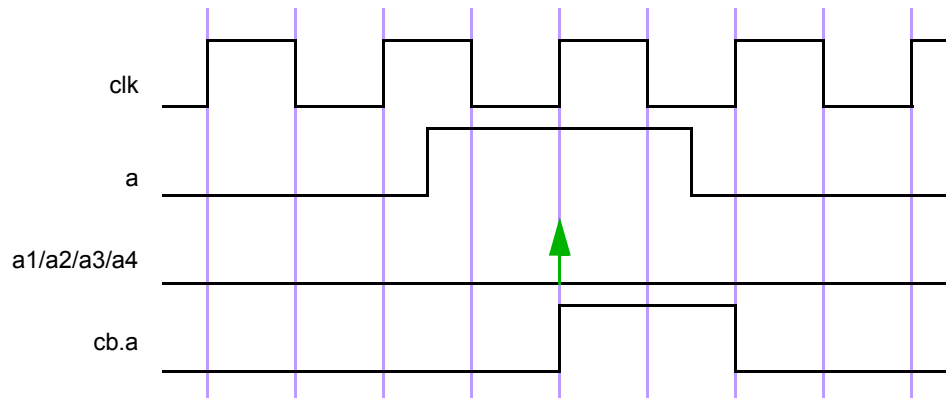Figure 17-16 explains the behavior of all the assertions. In the above example, a1, a2, a3, and a4 are equivalent.

Figure 17-16—Clocking blocks and concurrent assertion

18. Coverage

18.1 Introduction

NOTE—Functional verification comprises a large portion of the resources required to design and validate a complex system. Often, the validation must be comprehensive without redundant effort. To minimize wasted effort, coverage is used as a guide for directing verification resources by identifying tested and untested portions of the design.

Coverage is defined as the percentage of verification objectives that have been met. It is used as a metric for evaluating the progress of a verification project in order to reduce the number of simulation cycles spent in verifying a design.

Broadly speaking, there are two types of coverage metrics: those that can be automatically extracted from the design code, such as code coverage, and those that are user-specified in order to tie the verification environment to the design intent or functionality. The latter form is referred to as *functional coverage* and is the topic of this clause.

Functional coverage is a user-defined metric that measures how much of the design specification, as enumerated by features in the test plan, has been exercised. It can be used to measure whether interesting scenarios, corner cases, specification invariants, or other applicable design conditions—captured as features of the test plan—have been observed, validated, and tested.

The key aspects of functional coverage are as follows:

— It is user-specified and is not automatically inferred from the design.
— It is based on the design specification (i.e., its intent) and is thus independent of the actual design code or its structure.

Because it is fully specified by the user, functional coverage requires more up-front effort (someone has to write the coverage model). Functional coverage also requires a more structured approach to verification. Although functional coverage can shorten the overall verification effort and yield higher quality designs, its shortcomings can impede its adoption.

The SystemVerilog functional coverage extensions address these shortcomings by providing language constructs for easy specification of functional coverage models. This specification can be efficiently executed by the SystemVerilog simulation engine, thus enabling coverage data manipulation and analysis tools that speed up the development of high-quality tests. The improved set of tests can exercise more corner cases and required scenarios, without redundant work.

The SystemVerilog functional coverage constructs enable the following:

— Coverage of variables and expressions, as well as cross coverage between them
— Automatic as well as user-defined coverage bins
— Associate bins with sets of values, transitions, or cross products
— Filtering conditions at multiple levels
— Events and sequences to automatically trigger coverage sampling
— Procedural activation and query of coverage
— Optional directives to control and regulate coverage

18.2 Defining the coverage model: covergroup

The **covergroup** construct encapsulates the specification of a coverage model. Each **covergroup** specification can include the following components:

— A clocking event that synchronizes the sampling of coverage points
— A set of coverage points
— Cross coverage between coverage points
— Optional formal arguments
— Coverage options

The **covergroup** construct is a user-defined type. The type definition is written once, and multiple instances of that type can be created in different contexts. Similar to a class, once defined, a **covergroup** instance can be created via the **new()** operator. A **covergroup** can be defined in a package, module, program, interface, or class.

```
covergroup_declaration ::=                                              // from A.2.11
      covergroup covergroup_identifier [ ( [ tf_port_list ] ) ] [ coverage_event ] ;
          { coverage_spec_or_option }
      endgroup [ : covergroup_identifier ]
coverage_spec_or_option ::=
      {attribute_instance} coverage_spec
   | {attribute_instance} coverage_option ;
coverage_option ::=
      option.member_identifier = expression
   | type_option.member_identifier = expression
coverage_spec ::=
      cover_point
   | cover_cross
coverage_event ::=
      clocking_event
   | @@( block_event_expression )
block_event_expression ::=
      block_event_expression or block_event_expression
   | begin hierarchical_btf_identifier
   | end hierarchical_btf_identifier
hierarchical_btf_identifier ::=
      hierarchical_tf_identifier
   | hierarchical_block_identifier
   | hierarchical_identifier [ class_scope ] method_identifier
variable_decl_assignment ::=                                            // from A.2.4
      ...
   | [ covergroup_variable_identifier ] = new [ ( list_of_arguments ) ]15
```

15. It shall be legal to omit the covergroup_variable_identifier from a covergroup instantiation only if this implicit instantiation is within a class that has no other instantiation of the covergroup.

Syntax 18-1—Covergroup syntax (excerpt from Annex A)

The identifier associated with the **covergroup** declaration defines the name of the coverage model. Using this name, an arbitrary number of coverage model instances can be created. For example:

```
covergroup cg; ... endgroup
cg cg_inst = new;
```

The above example defines a **covergroup** named cg. An instance of cg is declared as cg_inst and created using the **new** operator.

A **covergroup** can specify an optional list of arguments. When the **covergroup** specifies a list of formal arguments, its instances must provide to the **new** operator all the actual arguments that are not defaulted. Actual arguments are evaluated when the **new** operator is executed. A **ref** argument allows a different variable to be sampled by each instance of a **covergroup**. Input arguments will not track value of their arguments; they will use the value passed to the **new** operator.

If a clocking event is specified, it defines the event at which coverage points are sampled. If the clocking event is omitted, users must procedurally trigger the coverage sampling. This is done via the built-in `sample()` method (see 18.7). Optionally, the **strobe** option can be used to modify the sampling behavior. When the strobe option is not set (the default), a coverage point is sampled the instant the clocking event takes place, as if the process triggering the event were to call the built-in `sample()` method. If the clocking event occurs multiple times in a time step, the coverage point will also be sampled multiple times. The strobe option (see 18.6.1) can be used to specify that coverage points are sampled in the Postponed region, thereby filtering multiple clocking events so that only one sample per time slot is taken. The **strobe** option only applies to the scheduling of samples triggered by a clocking event. It shall have no effect on procedural calls to the built-in `sample()` method.

As an alternative to a clocking event, a coverage group accepts a block event expression to indicate that the coverage sample is to be triggered by the start or the end of execution of a given named block, task, function, or class method. Block event expressions that specify the **begin** keyword followed by a hierarchical identifier denoting a named block, task, function, or class method shall be triggered immediately before the corresponding block, task, function, or method begins executing its first statement. Block event expressions that specify the **end** keyword followed by a hierarchical identifier denoting a named block, task, function, or class method shall be triggered immediately after the corresponding block, task, function, or method executes its last statement. Block event expressions that specify the end of execution shall not be triggered if the block, task, function, or method is disabled.

A **covergroup** can contain one or more coverage points. A coverage point can be a variable or an expression. Each coverage point includes a set of bins associated with its sampled values or its value transitions. The bins can be explicitly defined by the user or automatically created by the tool. Coverage points are discussed in detail in 18.4.

```
enum { red, green, blue } color;

covergroup g1 @(posedge clk);
   c: coverpoint color;
endgroup
```

The above example defines coverage group `g1` with a single coverage point associated with variable color. The value of the variable color is sampled at the indicated clocking event: the positive edge of signal `clk`. Because the coverage point does not explicitly define any bins, the tool automatically creates three bins, one for each possible value of the enumerated type. Automatic bins are described in 18.4.2.

A coverage group can also specify cross coverage between two or more coverage points or variables. Any combination of more than two variables or previously declared coverage points is allowed. For example:

```
enum { red, green, blue } color;
bit [3:0] pixel_adr, pixel_offset, pixel_hue;

covergroup g2 @(posedge clk);
   Hue: coverpoint pixel_hue;
   Offset: coverpoint pixel_offset;

   AxC: cross color, pixel_adr;    // cross 2 variables (implicitly declared
                                   // coverpoints)
   all: cross color, Hue, Offset;  // cross 1 variable and 2 coverpoints
endgroup
```

The example above creates coverage group `g2` that includes two coverage points and two cross coverage items. Explicit coverage points labeled `Offset` and `Hue` are defined for variables `pixel_offset` and

`pixel_hue`. SystemVerilog implicitly declares coverage points for variables `color` and `pixel_adr` in order to track their cross coverage. Implicitly declared cover points are described in 18.5.

A coverage group can also specify one or more options to control and regulate how coverage data are structured and collected. Coverage options can be specified for the coverage group as a whole or for specific items within the coverage group, that is, any of its coverage points or crosses. In general, a coverage option specified at the **covergroup** level applies to all of its items unless overridden by them. Coverage options are described in 18.6.

18.3 Using covergroup in classes

By embedding a coverage group within a class definition, the **covergroup** provides a simple way to cover a subset of the class properties. This integration of coverage with classes provides an intuitive and expressive mechanism for defining the coverage model associated with a class. For example:

In class `xyz`, defined below, members `m_x` and `m_y` are covered using an embedded **covergroup**:

```
class xyz;
    bit [3:0] m_x;
    int m_y;
    bit m_z;

    covergroup cov1 @m_z;          // embedded covergroup
        coverpoint m_x;
        coverpoint m_y;
    endgroup

    function new(); cov1 = new; endfunction
endclass
```

In this example, data members `m_x` and `m_y` of class `xyz` are sampled on every change of data member `m_z`.

When a **covergroup** is defined within a class and no explicit variables of that **covergroup** are declared in the class, then a variable with the same name as the coverage group is implicitly declared, e.g, in the above example, a variable `cov1` (of the embedded coverage group) is implicitly declared. Whether the coverage group variable is implicitly or explicitly declared, each class contains exactly one variable of each embedded coverage group. Each embedded coverage group thus becomes part of the class, tightly binding the class properties to the coverage definition. Declaring multiple variables of the same embedded coverage group shall result in a compiler error.

An embedded **covergroup** can define a coverage model for protected and local class properties without any changes to the class data encapsulation. Class members can become coverage points or can be used in other coverage constructs, such as conditional guards or option initialization.

A class can have more than one **covergroup**. The following example shows two coverage groups in class MC.

```
class MC;
    logic [3:0] m_x;
    local logic m_z;
    bit m_e;
    covergroup cv1 @(posedge clk); coverpoint m_x; endgroup
    covergroup cv2 @m_e ; coverpoint m_z; endgroup
endclass
```

306

In **covergroup** cv1, public class member variable m_x is sampled at every positive edge of signal clk. Local class member m_z is covered by another **covergroup** cv2. Each coverage group is sampled by a different clocking event.

An embedded coverage group must be explicitly instantiated in the **new** method. If it is not, then the coverage group is not created and no data will be sampled.

Below is an example of an embedded coverage group that does not have any passed-in arguments and uses explicit instantiation to synchronize with another object:

```
class Helper;
   int m_ev;
endclass

class MyClass;
   Helper m_obj;
   int m_a;
   covergroup Cov @(m_obj.m_ev);
      coverpoint m_a;
   endgroup

   function new();
      m_obj = new;

      Cov = new;      // Create embedded covergroup after creating m_obj
   endfunction
endclass
```

In this example, **covergroup** Cov is embedded within class MyClass, which contains an object of type Helper class, called m_obj. The clocking event for the embedded coverage group refers to data member m_ev of m_obj. Because the coverage group Cov uses m_obj, m_obj must be instantiated before Cov. Therefore, the coverage group Cov is instantiated after instantiating m_obj in the class constructor. As shown above, the instantiation of an embedded coverage group is done by assigning the result of the **new** operator to the coverage group identifier.

The following example shows how arguments passed in to an embedded coverage group can be used to set a coverage option of the coverage group:

```
class C1;
   bit [7:0] x;

   covergroup cv (int arg) @(posedge clk);
      option.at_least = arg;
      coverpoint x;
   endgroup

   function new(int p1);
      cv = new(p1);
   endfunction
endclass

initial begin
   C1 obj = new(4);
end
```

18.4 Defining coverage points

A **covergroup** can contain one or more coverage points. A coverage point can be an integral variable or an integral expression. Each coverage point includes a set of bins associated with its sampled values or its value transitions. The bins can be explicitly defined by the user or automatically created by SystemVerilog. The syntax for specifying coverage points is given below.

```
cover_point ::=                                                          // from A.2.11
      [ cover_point_identifier : ] coverpoint expression [ iff ( expression ) ] bins_or_empty
bins_or_empty ::=
       { {attribute_instance} { bins_or_options ; } }
      | ;
bins_or_options ::=
       coverage_option
      | [ wildcard ] bins_keyword bin_identifier [ [ [ expression ] ] ] = { open_range_list } [ iff ( expression ) ]
      | [ wildcard] bins_keyword bin_identifier [ [ ] ] = trans_list [ iff ( expression ) ]
      | bins_keyword bin_identifier [ [ [ expression ] ] ] = default [ iff ( expression ) ]
      | bins_keyword bin_identifier = default sequence [ iff ( expression ) ]
bins_keyword::= bins | illegal_bins | ignore_bins
open_range_list ::= open_value_range { , open_value_range }
open_value_range ::= value_range[20]
```

Syntax 18-2—Coverage point syntax (excerpt from Annex A)

A coverage point creates a hierarchical scope and can be optionally labeled. If the label is specified, then it designates the name of the coverage point. This name can be used to add this coverage point to a cross coverage specification or to access the methods of the coverage point. If the label is omitted and the coverage point is associated with a single variable, then the variable name becomes the name of the coverage point. Otherwise, an implementation can generate a name for the coverage point only for the purposes of coverage reporting, that is, generated names cannot be used within the language.

A coverage point can sample the values that correspond to a particular scheduling region (see Clause 9) by specifying a **clocking** block signal. Thus, a coverage point that denotes a **clocking** block signal will sample the values made available by the **clocking** block. If the **clocking** block specifies a skew of #1step, the coverage point will sample the signal values from the Preponed region. If the **clocking** block specifies a skew of #0, the coverage point will sample the signal values from the Observe region.

The expression within the **iff** construct specifies an optional condition that disables coverage for that cover point. If the guard expression evaluates to false at a sampling point, the coverage point is ignored. For example:

```
covergroup g4;
   coverpoint s0 iff(!reset);
endgroup
```

In the preceding example, cover point s0 is covered only if the value reset is false.

A coverage point bin associates a name and a count with a set of values or a sequence of value transitions. If the bin designates a set of values, the count is incremented every time the coverage point matches one of the values in the set. If the bin designates a sequence of value transitions, the count is incremented every time the coverage point matches the entire sequence of value transitions.

The bins for a coverage point can be automatically created by SystemVerilog or explicitly defined using the **bins** construct to name each bin. If the bins are not explicitly defined, they are automatically created by SystemVerilog. The number of automatically created bins can be controlled using the `auto_bin_max` coverage option. Coverage options are described in 18.6.

The **bins** construct allows creating a separate bin for each value in the given range list or a single bin for the entire range of values. To create a separate bin for each value (an array of bins), the square brackets, **[]**, must follow the bin name. To create a fixed number of bins for a set of values, a number can be specified inside the square brackets. The *open_range_list* used to specify the set of values associated with a bin shall be constant expressions, instance constants (for classes only), or non-**ref** arguments to the coverage group. It shall be legal to use the `$` primary in an *open_value_range* of the form `[ expression : $ ]` or `[ $ : expression ]`.

If a fixed number of bins is specified and that number is smaller than the specified number of values, then the possible bin values are uniformly distributed among the specified bins. The first 'n' specified values are assigned to the first bin, the next 'n' specified values are assigned to the next bin, etc. Duplicate values are retained; thus the same value can be assigned to multiple bins. If the number of values is not divisible by the number of bins, then the last bin will include the remaining items. For example:

```
bins fixed [4] = {1:10, 1, 4, 7};
```

The 13 possible values are distributed as follows: <1,2,3>, <4,5,6>, <7,8,9>, <10,1,4,7>. If the number of bins exceeds the number of values, then some of the bins will be empty.

The expression within the **iff** construct at the end of a bin definition provides a per-bin guard condition. If the expression is false at a sampling point, the count for the bin is not incremented.

The **default** specification defines a bin that is associated with none of the defined value bins. The **default** bin catches the values of the coverage point that do not lie within any of the defined bins. However, the coverage calculation for a coverage point shall not take into account the coverage captured by the default bin. The **default** bin is also excluded from cross coverage (see 18.5). The default is useful for catching unplanned or invalid values. The **default sequence** form can be used to catch all transitions (or sequences) that do not lie within any of the defined transition bins (see 18.4.1). The **default sequence** specification does not accept multiple transition bins (i.e., the `[]` notation is not allowed).

```
bit [9:0] v_a;

covergroup cg @(posedge clk);

    coverpoint v_a
    {
        bins a = { [0:63],65 };
        bins b[] = { [127:150],[148:191] };    // note overlapping values
        bins c[] = { 200,201,202 };
        bins d = { [1000:$] };
        bins others[] = default;
    }
endgroup
```

In the example above, the first **bins** construct associates bin `a` with the values of variable `v_a` between 0 and 63 and the value 65. The second **bins** construct creates a set of 65 bins `b[127]`, `b[128]`,...`b[191]`. Likewise, the third **bins** construct creates 3 bins: `c[200]`, `c[201]`, and `c[202]`. The fourth **bins** construct associates bin `d` with the values between 1000 and 1023 (`$` represents the maximum value of `v_a`). Every value that does not match bins `a`, `b[]`, `c[]`, or `d[]` is added into its own distinct bin.

A **default** or **default sequence** bin specification cannot be explicitly ignored (see 18.4.4). It shall be an error for bins designated as **ignore_bins** to also specify a **default** or **default sequence**.

Generic coverage groups can be written by passing their traits as arguments to the constructor. For example:

```
covergroup gc (ref int ra, int low, int high ) @(posedge clk);

    coverpoint ra  // sample variable passed by reference
    {
       bins good = { [low : high] };
       bins bad[] = default;
    }
endgroup


...
int va, vb;

cg c1 = new( va, 0, 50 );    // cover variable va in the range 0 to 50
cg c2 = new( vb, 120, 600 ); // cover variable vb in the range 120 to 600
```

The example above defines a coverage group, gc, in which the signal to be sampled and the extent of the coverage bins are specified as arguments. Later, two instances of the coverage group are created; each instance samples a different signal and covers a different range of values.

18.4.1 Specifying bins for transitions

The syntax for specifying transition bins accepts a subset of the sequence syntax described in Clause 17:

bins_or_options ::=   // from A.2.11
    ...
   | [**wildcard**] bins_keyword bin_identifier [**[** [expression] **]**] **= {** open_range_list **}** [**iff (** expression **)**]
   | [**wildcard**] bins_keyword bin_identifier [**[ ]**] **=** trans_list [**iff (** expression **)**]
    ...
bins_keyword::= **bins** | **illegal_bins** | **ignore_bins**
open_range_list ::= open_value_range { **,** open_value_range }
trans_list ::= **(** trans_set **)** { **,** **(** trans_set **)** }
trans_set ::= trans_range_list { **=>** trans_range_list }
trans_range_list ::=
    trans_item
   | trans_item [**[*** repeat_range **]**]
   | trans_item [**[–>** repeat_range **]**]
   | trans_item [**[=** repeat_range **]**]
trans_item ::= range_list
repeat_range ::=
    expression
   | expression **:** expression

Syntax 18-3—Transition bin syntax (excerpt from Annex A)

A *trans_list* specifies one or more sets of ordered value transitions of the coverage point. A single value transition is thus specified as follows:

```
value1 => value2
```

It represents the value of coverage point at two successive sample points, that is, `value1` followed by `value2` at the next sample point.

A sequence of transitions is represented as follows:

```
value1 => value3 => value4 => value5
```

In this case, `value1` is followed by `value3`, followed by `value4`, and followed by `value5`. A sequence can be of any arbitrary length.

A set of transitions can be specified as follows:

```
range_list1 => range_list2
```

This specification expands to transitions between each value in `range_list1` and each value in `range_list2`. For example:

```
1,5 => 6, 7
```

specifies the following four transitions:

```
( 1=>6 ), ( 1=>7 ), ( 5=>6 ), ( 5=>7 )
```

Consecutive repetitions of transitions are specified using (see Annex E):

```
trans_item [* repeat_range ]
```

Here, *trans_item* is repeated for *repeat_range* times. For example:

```
3 [* 5]
```

is the same as

```
3=>3=>3=>3=>3
```

An example of a range of repetition is as follows:

```
3 [* 3:5]
```

which is the same as

```
( 3=>3=>3 ), ( 3=>3=>3=>3 ), ( 3=>3=>3=>3=>3 )
```

The repetition with nonconsecutive occurrence of a value is specified using: trans_item `[-> `repeat_range` ]`. Here, the occurrence of a value is specified with an arbitrary number of sample points where the value does not occur. For example:

```
3 [-> 3]
```

is the same as

```
...3=>...=>3...=>3
```

where the dots (. . .) represent any transition that does not contain the value 3.

Nonconsecutive repetition is where a sequence of transitions continues until the next transition. For example:

```
3 [= 2]
```

is same as the transitions below excluding the last transition.

```
3=>...=>3...=>3
```

A *trans_list* specifies one or more sets of ordered value transitions of the coverage point. If the sequence of value transitions of the coverage point matches any complete sequence in the *trans_list*, the coverage count of the corresponding bin is incremented. For example:

```
bit [4:1] v_a;

covergroup cg @(posedge clk);

   coverpoint v_a
   {
      bins sa = (4 => 5 => 6), ([7:9],10=>11,12);
      bins sb[] = (4=> 5 => 6), ([7:9],10=>11,12);
      bins allother = default sequence ;
   }
endgroup
```

The example above defines two transition coverage bins. The first **bins** construct associates the following sequences with bin `sa`: `4=>5=>6`, or `7=>11, 8=>11, 9=>11, 10=>11, 7=>12, 8=>12, 9=>12, 10=>12`. The second **bins** construct associates an individual bin with each of the above sequences: `sb[4=>5=>6]`, `...`, `sb[10=>12]`. The bin `allother` tracks all other transitions that are not covered by the other bins: `sa` and `sb`.

Transitions that specify sequences of unbounded or undetermined varying length cannot be used with the multiple bins construct (the `[]` notation). For example, the length of the transition `3[=2]`, which uses nonconsecutive repetition, is unbounded and can vary during simulation. An attempt to specify multiple bins with such sequences shall result in an error.

A transition bin is incremented every time the sequence of value transitions of its corresponding coverage point matches a complete sequence, even when the sequences overlap. For example, given the definition

```
covergroup sg @(posedge clk);
   coverpoint v
   {
      bins b2 = ( 2 [-> 3:5] );    // 3 to 5 nonconsecutive 2's
      bins b3 = ( 3 [-> 3:5] );    // 3 to 5 nonconsecutive 3's
      bins b5 = ( 5 [* 3] );       // 3 consecutive 5's
   }
endgroup
```

and the sequence of sampled values for coverpoint variable `v`

```
1 2 3 2 3 2 3 2 3 2 3 1 5 5 5 5 5 5
```

the above sequence causes transition bin `b2` to be incremented on the 6th sample (3 nonconsecutive twos), and transition bin `b3` to be incremented on the 7th sample (3 nonconsecutive threes). Likewise, transition bin `b2` is incremented on the 8th and 10th samples, and transition bin `b3` is incremented on the 9th and 11th samples. Next, transition bin `b5` is incremented the on the 15th, 16th, 17th, and 18th samples.

A transition bin is incremented at most once per sample. In the preceding example, on the 10th sample, the transition bin `b2` is incremented only once (1 is added to the bin count).

18.4.2 Automatic bin creation for coverage points

If a coverage point does not define any bins, SystemVerilog automatically creates state bins. This provides an easy-to-use mechanism for binning different values of a coverage point. Users can either let the tool automatically create state bins for coverage points or explicitly define named bins for each coverage point.

When the automatic bin creation mechanism is used, SystemVerilog creates N bins to collect the sampled values of a coverage point. The value N is determined as follows:

— For an **enum** coverage point, N is the cardinality of the enumeration.

— For any other integral coverage point, N is the minimum of 2^M and the value of the `auto_bin_max` option, where M is the number of bits needed to represent the coverage point.

If the number of automatic bins is smaller than the number of possible values ($N < 2^M$), then the 2^M values are uniformly distributed in the N bins. If the number of values, 2^M, is not divisible by N, then the last bin will include the additional (up to $N-1$) remaining items. For example, if M is 3 and N is 3, then the eight possible values are distributed as follows: <0:1>, <2:3>, <4,5,6,7>.

Automatically created bins only consider 2-state values; sampled values containing **x** or **z** are excluded.

SystemVerilog implementations can impose a limit on the number of automatic bins. See the 18.6 for the default value of `auto_bin_max`.

Each automatically created bin will have a name of the form of `auto[value]` where `value` is either a single coverage point value or the range of coverage point values included in the bin—in the form low**:**high. For enumerated types, `value` is the named constant associated with a particular enumerated value.

18.4.3 Wildcard specification of coverage point bins

By default, a value or transition bin definition can specify 4-state values. When a bin definition includes an **x** or **z**, it indicates that the bin count should only be incremented when the sampled value has an **x** or **z** in the same bit positions, i.e., the comparison is done using `===`. The **wildcard bins** definition causes all **X**, **Z**, or **?** to be treated as wildcards for **0** or **1** (similar to the `==?` operator). For example:

```
wildcard bins g12_16 = { 4'b11?? };
```

The count of bin `g12_16` is incremented when the sampled variable is between 12 and 16:

```
1100    1101    1110    1111
```

Similarly, transition bins can define **wildcard bins**. For example:

```
wildcard bins T0_3 = (2'b0x => 2'b1x);
```

The count of transition bin `T0_3` is incremented for the following transitions (as if by `(0,1=>2,3)`):

```
00 => 10    00 => 11    01 => 10    01 => 11
```

A wildcard bin definition only considers 2-state values; sampled values containing **x** or **z** are excluded. Thus, the range of values covered by a wildcard bin is established by replacing every wildcard digit by 0 to compute the low bound and 1 to compute the high bound.

18.4.4 Excluding coverage point values or transitions

A set of values or transitions associated with a coverage point can be explicitly excluded from coverage by specifying them as `ignore_bins`. For example:

```
covergroup cg23;
   coverpoint a
   {
      ignore_bins ignore_vals = {7,8};
      ignore_bins ignore_trans = (1=>3=>5);
   }
endgroup
```

All values or transitions associated with ignored bins are excluded from coverage. Ignored values or transitions are excluded even if they are also included in another bin.

18.4.5 Specifying Illegal coverage point values or transitions

A set of values or transitions associated with a coverage point can be marked as illegal by specifying them as `illegal_bins`. For example:

```
covergroup cg3;
   coverpoint b
   {
      illegal_bins bad_vals = {1,2,3};
      illegal_bins bad_trans = (4=>5=>6);
   }
endgroup
```

All values or transitions associated with illegal bins are excluded from coverage. If they occur, a run-time error is issued. Illegal bins take precedence over any other bins, that is, they will result in a run-time error even if they are also included in another bin.

18.5 Defining cross coverage

A coverage group can specify cross coverage between two or more coverage points or variables. Cross coverage is specified using the `cross` construct. When a variable `v` is part of a cross coverage, SystemVerilog implicitly creates a coverage point for the variable, as if it had been created by the statement `coverpoint v;`. Thus, a cross involves only coverage points. Expressions cannot be used directly in a cross; a coverage point must be explicitly defined first.

The syntax for specifying cross coverage is given below.

```
cover_cross ::=                                                          // from A.2.11
        [ cover_point_identifier : ] cross list_of_coverpoints [ iff ( expression ) ] select_bins_or_empty
list_of_coverpoints ::= cross_item , cross_item { , cross_item }
cross_item ::=
        cover_point_identifier
    | variable_identifier
select_bins_or_empty ::=
        { { bins_selection_or_option ; } }
    | ;
bins_selection_or_option ::=
        { attribute_instance } coverage_option
    | { attribute_instance } bins_selection
bins_selection ::= bins_keyword bin_identifier = select_expression [ iff ( expression ) ]
select_expression ::=
        select_condition
    | ! select_condition
    | select_expression && select_expression
    | select_expression || select_expression
    | ( select_expression )
select_condition ::= binsof ( bins_expression ) [ intersect { open_range_list } ]
bins_expression ::=
        variable_identifier
    | cover_point_identifier [ . bins_identifier ]
open_range_list ::= open_value_range { , open_value_range }
open_value_range ::= value_range[20]
```

Syntax 18-4—Cross coverage syntax (excerpt from Annex A)

The label for a **cross** declaration provides an optional name. The label also creates a hierarchical scope for the **bins** defined within the **cross**.

The expression within the optional **iff** provides a conditional guard for the cross coverage. If at any sample point, the condition evaluates to false, the cross coverage is ignored. The expression within the optional **iff** construct at the end of a cross bin definition provides a per-bin guard condition. If the expression is false, the cross bin is ignored.

Cross coverage of a set of *N* coverage points is defined as the coverage of all combinations of all bins associated with the *N* coverage points, that is, the Cartesian product of the *N* sets of coverage point bins. For example:

```
bit [3:0] a, b;

covergroup cov @(posedge clk);
    aXb : cross a, b;
endgroup
```

The coverage group cov in the example above specifies the cross coverage of two 4-bit variables, a and b. SystemVerilog implicitly creates a coverage point for each variable. Each coverage point has 16 bins, namely auto[0]...auto[15]. The cross of a and b (labeled aXb), therefore, has 256 cross products, and each cross product is a bin of aXb.

Cross coverage between expressions previously defined as coverage points is also allowed. For example:

```
bit [3:0] a, b, c;

covergroup cov2 @(posedge clk);
    BC: coverpoint b+c;
    aXb : cross a, BC;
endgroup
```

The coverage group `cov2` has the same number of cross products as the previous example, but in this case, one of the coverage points is the expression `b+c`, which is labeled `BC`.

```
bit [31:0] a_var;
bit [3:0] b_var;

covergroup cov3 @(posedge clk);
    A: coverpoint a_var { bins yy[] = { [0:9] }; }
    CC: cross b_var, A;
endgroup
```

The coverage group `cov3` crosses variable `b_var` with coverage point `A` (labeled `CC`). Variable `b_var` automatically creates 16 bins (`auto[0]...auto[15]`). Coverage point `A` explicitly creates 10 bins (`yy[0]...yy[9]`). The cross of two coverage points creates 16 * 10 = 160 cross product bins, namely the pairs shown below:

```
<auto[0], yy[0]>
<auto[0], yy[1]>
...
<auto[0], yy[9]>
<auto[1], yy[0]>
...
<auto[15], yy[9]>
```

No cross coverage bins shall be created for coverpoint bins that are specified as **default**, ignored, or illegal bins.

Cross coverage is allowed only between coverage points defined within the same coverage group. Coverage points defined in a coverage group other than the one enclosing the cross cannot participate in a cross. Attempts to cross items from different coverage groups shall result in a compiler error.

In addition to specifying the coverage points that are crossed, SystemVerilog includes a powerful set of operators that allow defining cross coverage bins. Cross coverage bins can be specified in order to group together a set of cross products. A cross coverage bin associates a name and a count with a set of cross products. The count of the bin is incremented every time any of the cross products match, i.e., every coverage point in the cross matches its corresponding bin in the cross product.

User-defined bins for cross coverage are defined using bins select expressions. The syntax for defining these bins select expressions is given in Syntax 18-4.

The **binsof** construct yields the bins of its expression, which can be either a coverage point (explicitly defined or implicitly defined for a single variable) or a coverage point bin. The resulting bins can be further selected by including (or excluding) only the bins whose associated values intersect a desired set of values. The desired set of values can be specified using a comma-separated list of `open_value_range` as shown in Syntax 18-4. For example, the select expression

```
binsof( x ) intersect { y }
```

denotes the bins of coverage point `x` whose values intersect the range given by `y`. Its negated form

```
! binsof( x ) intersect { y }
```

denotes the bins of coverage point `x` whose values do not intersect the range given by `y`.

The `open_value_range` syntax can specify a single value, a range of values, or an open range, which denotes the following:

```
[ $ : value ] => The set of values less than or equal to value
[ value : $ ] => The set of values greater or equal to value
```

The bins selected can be combined with other selected bins using the logical operators **&&** and **||** .

18.5.1 Example of user-defined cross coverage and select expressions

```
bit [7:0] v_a, v_b;

covergroup cg @(posedge clk);

    a: coverpoint v_a
    {
        bins a1 = { [0:63] };
        bins a2 = { [64:127] };
        bins a3 = { [128:191] };
        bins a4 = { [192:255] };
    }

    b: coverpoint v_b
    {
        bins b1 = {0};
        bins b2 = { [1:84] };
        bins b3 = { [85:169] };
        bins b4 = { [170:255] };
    }

    c : cross a, b
    {
        bins c1 = ! binsof(a) intersect {[100:200]};// 4 cross products
        bins c2 = binsof(a.a2) || binsof(b.b2);// 7 cross products
        bins c3 = binsof(a.a1) && binsof(b.b4);// 1 cross product
    }
endgroup
```

The example above defines a coverage group named `cg` that samples its coverage points on the positive edge of signal `clk` (not shown). The coverage group includes two coverage points, one for each of the two 8-bit variables, a and b. Coverage point a associated with variable `v_a` defines four equal-sized bins for each possible value of variable `v_a`. Likewise, coverage point b associated with variable `v_b` defines four bins for each possible value of variable `v_b`. Cross definition c specifies the cross coverage of the two coverage points `v_a` and `v_b`. If the cross coverage of coverage points a and b were defined without any additional cross bins (select expressions), then cross coverage of a and b would include 16 cross products corresponding to all combinations of bins a1 through a4 with bins b1 through b4, that is, cross products <a1,b1>, <a1,b2>, <a1,b3>, <a1,b4>. . .<a4, b1>, <a4,b2>, <a4,b3>, <a4,b4>.

The first user-defined cross bin, `c1`, specifies that `c1` should include only cross products of coverage point `a` that do not intersect the value range of 100 to 200. This select expression excludes bins `a2`, `a3`, and `a4`. Thus, `c1` will cover only four cross products of <a1,b1>, <a1,b2>, <a1,b3>, and <a1,b4>.

The second user-defined cross bin, `c2`, specifies that bin `c2` should include only cross products whose values are covered by bin `a2` of coverage point `a` or cross products whose values are covered by bin `b2` of coverage point `b`. This select expression includes the following seven cross products: <a2,b1>, <a2,b2>, <a2,b3>, <a2,b4>, <a1,b2>, <a3,b2>, and <a4,b2>.

The final user-defined cross bin, `c3`, specifies that `c3` should include only cross products whose values are covered by bin `a1` of coverage point `a` and cross products whose values are covered by bin `b4` of coverage point `b`. This select expression includes only one cross product: <a1, b4>.

When select expressions are specified on transition bins, the **binsof** operator uses the last value of the transition.

18.5.2 Excluding cross products

A group of bins can be excluded from coverage by specifying a select expression using **ignore_bins**. For example:

```
covergroup yy;
  cross a, b
  {
    ignore_bins foo = binsof(a) intersect { 5, [1:3] };
  }
endgroup
```

All cross products that satisfy the select expression are excluded from coverage. Ignored cross products are excluded even if they are included in other cross coverage bins of the enclosing cross.

18.5.3 Specifying Illegal cross products

A group of bins can be marked as illegal by specifying a select expression using **illegal_bins**. For example:

```
covergroup zz(int bad);
  cross x, y
  {
    illegal_bins foo = binsof(y) intersect {bad};
  }
endgroup
```

All cross products that satisfy the select expression are excluded from coverage, and a run-time error is issued. Illegal cross products take precedence over any other cross products, that is, they will result in a run-time error even if they are also explicitly ignored (using an **ignore_bins**) or included in another cross bin.

18.6 Specifying coverage options

Options control the behavior of the **covergroup**, **coverpoint**, and **cross**. There are two types of options: those that are specific to an instance of a **covergroup** and those that specify an option for the **covergroup** type as a whole.

Specifying a value for the same option more than once within the same **covergroup** definition shall be an error.

Table 18-1 lists instance-specific **covergroup** options and their description. Each instance of a **covergroup** can initialize an instance-specific option to a different value. The initialized option value affects only that instance.

Table 18-1— Instance-specific coverage options

Option name	Default	Description
weight= *number*	1	If set at the **covergroup** syntactic level, it specifies the weight of this **covergroup** instance for computing the overall instance coverage of the simulation. If set at the **coverpoint** (or **cross**) syntactic level, it specifies the weight of a coverpoint (or **cross**) for computing the instance coverage of the enclosing **covergroup.**
goal=*number*	90	Specifies the target goal for a **covergroup** instance or for a **coverpoint** or a **cross** of an instance.
name=*string*	unique name	Specifies a name for the **covergroup** instance. If unspecified, a unique name for each instance is automatically generated by the tool.
comment=*string*	""	A comment that appears with a **covergroup** instance or with a **coverpoint** or **cross** of the **covergroup** instance. The comment is saved in the coverage database and included in the coverage report.
at_least=*number*	1	Minimum number of hits for each bin. A bin with a hit count that is less than *number* is not considered covered.
detect_overlap=*boolean*	0	When true, a warning is issued if there is an overlap between the range list (or transition list) of two bins of a **coverpoint**.
auto_bin_max=*number*	64	Maximum number of automatically created bins when no bins are explicitly defined for a **coverpoint**.
cross_num_print_missing= *number*	0	Number of missing (not covered) cross product bins that must be saved to the coverage database and printed in the coverage report.
per_instance=*boolean*	0	Each instance contributes to the overall coverage information for the **covergroup** type. When true, coverage information for this **covergroup** instance is tracked as well.

The instance-specific options mentioned above can be set in the **covergroup** definition. The syntax for setting these options in the **covergroup** definition is as follows:

```
option.option_name = expression ;
```

The identifier **option** is a built-in member of any coverage group (see 18.9 for a description).

For example:

```
covergroup g1 (int w, string instComment) @(posedge clk) ;
   // track coverage information for each instance of g1 in addition
   // to the cumulative coverage information for covergroup type g1
   option.per_instance = 1;
```

```
      // comment for each instance of this covergroup
      option.comment = instComment;

      a : coverpoint a_var
      {
         // Create 128 automatic bins for coverpoint "a" of each instance of g1
         option.auto_bin_max = 128;
      }
      b : coverpoint b_var
      {
         // This coverpoint contributes w times as much to the coverage of an
         // instance of g1 as coverpoints "a" and "c1"
         option.weight = w;
         }
      c1 : cross a_var, b_var ;
   endgroup
```

Option assignment statements in the **covergroup** definition are evaluated at the time that the **covergroup** is instantiated. The **per_instance** option can only be set in the **covergroup** definition. Other instance-specific options can be set procedurally after a **covergroup** has been instantiated. The syntax is as follows:

coverage_option_assignment ::=   *// not in Annex A*
      instance_name.**option**.option_name **=** expression **;**
    | instance_name.covergroup_item_identifier.**option.**option_name **=** expression **;**

Syntax 18-5—Coverage option assignment syntax (not in Annex A)

For example:

```
   covergroup gc @(posedge clk) ;
      a : coverpoint a_var;
      b : coverpoint b_var;
   endgroup
   ...
   gc g1 = new;
   g1.option.comment = "Here is a comment set for the instance g1";
   g1.a.option.weight = 3; // Set weight for coverpoint "a" of instance g1
```

Table 18-2 summarizes the syntactical level (**covergroup**, **coverpoint**, or **cross**) at which instance options can be specified. All instance options can be specified at the **covergroup** level. Except for the **weight**, **goal**, **comment**, and **per_instance** options, all other options set at the **covergroup** syntactic level act as a default value for the corresponding option of all coverpoints and crosses in the **covergroup**. Individual coverpoints or crosses can overwrite these default values. When set at the **covergroup** level, the **weight**, **goal**, **comment**, and **per_instance** options do not act as default values to the lower syntactic levels.

Table 18-2—Coverage options per-syntactic level

Option name	Allowed in Syntactic Level		
	covergroup	coverpoint	cross
name	Yes	No	No
weight	Yes	Yes	Yes
goal	Yes	Yes	Yes
comment	Yes	Yes	Yes
at_least	Yes (default for coverpoints & crosses)	Yes	Yes
detect_overlap	Yes (default for coverpoints)	Yes	No
auto_bin_max	Yes (default for coverpoints)	Yes	No
cross_num_print_missing	Yes (default for crosses)	No	Yes
per_instance	Yes	No	No

18.6.1 Covergroup type options

Table 18-3 lists options that describe particular features (or properties) of the **covergroup** type as a whole. They are analogous to static data members of classes.

Table 18-3—Coverage group type (static) options

Option name	Default	Description
weight=*constant_number*	1	If set at the **covergroup** syntactic level, it specifies the weight of this **covergroup** for computing the overall cumulative (or type) coverage of the saved database. If set at the **coverpoint** (or **cross**) syntactic level, it specifies the weight of a **coverpoint** (or **cross**) for computing the cumulative (or type) coverage of the enclosing **covergroup.**
goal=*constant_number*	100	Specifies the target goal for a **covergroup** type or for a **coverpoint** or **cross** of a **covergroup** type.
comment=*string_literal*	""	A comment that appears with the **covergroup** type or with a **coverpoint** or **cross** of the **covergroup** type. The comment is saved in the coverage database and included in the coverage report.
strobe=*boolean*	0	When true, all samples happen at the end of the time slot, like the `$strobe` system task.

The **covergroup** type options mentioned above can be set in the **covergroup** definition. The syntax for setting these options in the **covergroup** definition is as follows:

```
type_option.option_name = expression ;
```

The identifier **type_option** is a built-in member of any coverage group (see 18.9 for a description).

Different instances of a **covergroup** cannot assign different values to type options. This is syntactically disallowed because these options can only be initialized via constant expressions. For example:

```
covergroup g1 (int w, string instComment) @(posedge clk) ;
   // track coverage information for each instance of g1 in addition
   // to the cumulative coverage information for covergroup type g1
   option.per_instance = 1;

   type_option.comment = "Coverage model for features foo and bar";

   type_option.strobe = 1;    // sample at the end of the time slot

   // comment for each instance of this covergroup
   option.comment = instComment;

   a : coverpoint a_var
   {
      // Use weight 2 to compute the coverage of each instance
      option.weight = 2;
      // Use weight 3 to compute the cumulative (type) coverage for g1
      type_option.weight = 3;
      // NOTE: type_option.weight = w would cause syntax error.
   }
   b : coverpoint b_var
   {
      // Use weight w to compute the coverage of each instance
      option.weight = w;
      // Use weight 5 to compute the cumulative (type) coverage of g1
      type_option.weight = 5;
   }
endgroup
```

In the above example, the coverage for each instance of g1 is computed as follows:

(((instance coverage of "a") * 2) + ((instance coverage of "b") * w)) / (2 + w)

On the other hand, the coverage for **covergroup** type "g1" is computed as follows:

((((overall type coverage of "a") * 3) + ((overall type coverage of "b") * 5)) / (3 + 5).

The strobe type option can only be set in the **covergroup** definition. Other type options can be set procedurally at any time during simulation. The syntax is as follows:

---

coverage_type_option_assignment ::=   *// not in Annex A*
      covergroup_name**::**type_option.option_name **=** expression **;**
   | covergroup_name**::**covergroup_item_identifier**::type_option.**option_name **=** expression **;**

Syntax 18-6—Coverage type option assignment syntax (not in Annex A)

For example:

```
covergroup gc @(posedge clk) ;
   a : coverpoint a_var;
   b : coverpoint b_var;
endgroup
...
```

```
gc::type_option.comment = "Here is a comment for covergroup g1";

// Set the weight for coverpoint "a" of covergroup g1
gc::a::type_option.weight = 3;
gc g1 = new;
```

Table 18-4 summarizes the syntactical level (**covergroup**, **coverpoint**, or **cross**) in which type options can be specified. When set at the **covergroup** level, the type options do not act as defaults for lower syntactic levels.

Table 18-4—Coverage type options

Option name	Allowed syntactic level		
	covergroup	coverpoint	cross
weight	Yes	Yes	Yes
goal	Yes	Yes	Yes
comment	Yes	Yes	Yes
strobe	Yes	No	No

18.7 Predefined coverage methods

The coverage methods in Table 18-5 are provided for the **covergroup**. These methods can be invoked procedurally at any time.

Table 18-5—Predefined coverage methods

Method (function)	Can be called on			Description
	covergroup	coverpoint	cross	
void sample()	Yes	No	No	Triggers sampling of the **covergroup**
real get_coverage() **real** get_coverage(**ref int**, **ref int**)	Yes	Yes	Yes	Calculates type coverage number (0...100)
real get_inst_coverage() **real** get_inst_coverage(**ref int**, **ref int**)	Yes	Yes	Yes	Calculates the coverage number (0...100)
void set_inst_name(**string**)	Yes	No	No	Sets the instance name to the given string
void start()	Yes	Yes	Yes	Starts collecting coverage information
void stop()	Yes	Yes	Yes	Stops collecting coverage information

The get_coverage() method returns the cumulative (or type) coverage, which considers the contribution of all instances of a particular coverage item; and it is a static method that is available on both types (via the

`::` operator) and instances (using the `.` operator). In contrast, the `get_inst_coverage()` method returns the coverage of the specific instance on which it is invoked; thus, it can only be invoked via the `.` operator.

The `get_coverage()` and `get_inst_coverage()` methods both accept an optional set of arguments, a pair of **int** values passed by reference. When the optional arguments are specified, the `get_coverage()` and `get_inst_coverage()` methods assign to the first argument the value of the covered bins and to the second argument the number of bins for the given coverage item. These two values correspond to the numerator and the denominator used for calculating the particular coverage number (i.e., the return value before scaling by 100).

18.8 Predefined coverage system tasks and functions

SystemVerilog provides the following system tasks and functions to help manage coverage data collection.

— **$set_coverage_db_name** (name) sets the filename of the coverage database into which coverage information is saved at the end of a simulation run.

— **$load_coverage_db** (name) loads from the given filename the cumulative coverage information for all coverage group types.

— **$get_coverage** ( ) returns as a real number in the range of 0 to 100 the overall coverage of all coverage group types. This number is computed as described above.

18.9 Organization of option and type_option members

The type and type_option members of a **covergroup**, **coverpoint**, and **cross** are implicitly declared structures with the following composition:

```
struct           // covergroup option declaration
{
   string   name ;
   int      weight ;
   int      goal ;
   string   comment ;
   int      at_least ;
   int      auto_bin_max ;
   int      cross_num_print_missing ;
   bit      detect_overlap ;
   bit      per_instance ;
} option;

struct           // coverpoint option declaration
{
   int      weight ;
   int      goal ;
   string   comment ;
   int      at_least ;
   int      auto_bin_max ;
   bit      detect_overlap ;
} option;

struct           // cross option declaration
{
   int      weight ;
   int      goal ;
   string   comment ;
   int      at_least ;
```

```
   int      cross_num_print_missing ;
} option;

struct           // covergroup type_option declaration
{
   int      weight ;
   int      goal ;
   string   comment ;
   bit      strobe ;
} type_option;

struct           // coverpoint and cross type_option declaration
{
   int      weight ;
   int      goal ;
   string   comment ;
} type_option;
```

18.10 Coverage computation

This subclause describes how SystemVerilog computes functional coverage numbers. The cumulative (or type) coverage considers the contribution of all instances of a particular coverage item, and it is the value returned by the `get_coverage()` method. Thus, when applied to a **covergroup**, the `get_coverage()` method returns the contribution of all instances of that particular **covergroup**. In contrast, the `get_inst_coverage()` method returns the coverage of the specific coverage instance on which it is invoked. Because `get_coverage()` is a static method, it is available for both types (via the **::** operator) and instances (using the **.** operator).

The coverage of a coverage group, Cg , is the weighted average of the coverage of all items defined in the coverage group, and it is computed according to the following formulae:

$$C_g = \frac{\sum_i W_i * C_i}{\sum_i W_i}$$

where

> i ∈ set of coverage items (coverage points and crosses) defined in the coverage group
>
> W_i is the weight associated with item i
>
> C_i is the coverage of item i

The coverage of each item, C_i, is a measure of how much the item has been covered, and its computation depends on the type of coverage item: **coverpoint** or **cross**. Each of these is described in 18.10.1 and 18.10.2, respectively.

18.10.1 Coverpoint coverage computation

Coverage of a **coverpoint** item is computed differently depending on whether the bins of the coverage point are explicitly defined by the user (see 18.4.1) or automatically created by the tool (see 18.4.2). For user-defined bins, the coverage of a **coverpoint** is computed as follows:

$$C_i = \frac{|bins_{\text{cov } ered}|}{|bins|}$$

where

    $|bins|$ is the cardinality of the set of bins defined

    $|bins_{\text{cov } ered}|$ is the cardinality of the covered bins—the subset of all (defined) bins that are covered

For automatically generated bins, the coverage of a `coverpoint` is computed as follows:

$$C_i = \frac{|bins_{\text{cov } ered}|}{\text{MIN}(auto\_bin\_max, 2^M)}$$

where

    $|bins_{\text{cov } ered}|$ is the cardinality of the covered bins—the subset of all (auto-defined) bins that are covered

    M is the minimum number of bits needed to represent the `coverpoint`

    auto_bin_max   is the value of the `auto_bin_max` option in effect (see 18.6)

It is important to understand that the cumulative coverage considers the union of all significant bins; thus, it includes the contribution of all bins (including overlapping bins) of all instances.

To determine whether a particular bin of a coverage group is covered, the cumulative coverage computation considers the value of the `at_least` option of all instances being accumulated. Consequently, a bin is not considered covered unless its hit count equals or exceeds the maximum of all the `at_least` values of all instances. Use of the maximum represents the more conservative choice.

18.10.2 Cross coverage computation

The coverage of a `cross` item is computed according to the following formulae:

$$C_i = \frac{|bins_{\text{cov } ered}|}{B_c + B_u}$$

$$B_c = \left(\prod_j B_j\right) - B_b$$

where

    j $\in$ set of coverpoints being crossed

    B_j is the cardinality (number of bins) of the jth coverpoint being crossed

    B_c is the number of auto-cross bins

    B_u is the number of significant user-defined cross bins—excluding `ignore_bins` and `illegal_bins`

    B_b is the number of cross products that comprise all user-defined cross bins

The term B_u represents user-defined bins that contribute towards coverage.

19. Hierarchy

19.1 Introduction

NOTE—Verilog has a simple organization. All data, functions, and tasks are in modules except for system tasks and functions, which are global and can be defined in the PLI. A Verilog module can contain instances of other modules. Any uninstantiated module is at the top level. This does not apply to libraries, which, therefore, have a different status and a different procedure for analyzing them. A hierarchical name can be used to specify any named object from anywhere in the instance hierarchy. The module hierarchy is often arbitrary, and a lot of effort is spent in maintaining port lists.

In Verilog, only net, **reg**, **integer**, and **time** data types can be passed through module ports.

SystemVerilog adds many enhancements for representing design hierarchy:

— Packages containing declarations such as data, types, classes, tasks, and functions
— Separate compilation support
— A compilation-unit scope visible only within a compilation unit
— Nested module declarations to aid in representing self-contained models and libraries
— Relaxed rules on port declarations
— Simplified named port connections, using `.name`
— Implicit port connections, using **.***
— Time unit and time precision specifications bound to modules
— A concept of interfaces to bundle connections between modules (presented in Clause 20)

An important enhancement in SystemVerilog is the ability to pass a value of any data type through module ports, using nets or variables. This includes reals, arrays, and structures.

19.2 Packages

SystemVerilog packages provide an additional mechanism for sharing parameters, data, type, task, function, sequence, and property declarations among multiple SystemVerilog modules, interfaces, and programs. Packages are explicitly named scopes appearing at the outermost level of the source text (at the same level as top-level modules and primitives). Types, variables, tasks, functions, sequences, and properties may be declared within a package. Such declarations may be referenced within modules, macromodules, interfaces, programs, and other packages by either import or fully resolved name.

Packages must not contain any processes. Therefore, wire declarations with implicit continuous assignments are not allowed.

```
package_declaration ::=                                                    // from A.1.2
      { attribute_instance } package package_identifier ;
           [ timeunits_declaration ] { { attribute_instance } package_item }
      endpackage [ : package_identifier ]
package_item ::=                                                            // from A.1.10
      package_or_generate_item_declaration
    | anonymous_program
    | timeunits_declaration17
package_or_generate_item_declaration ::=
      net_declaration
    | data_declaration
    | task_declaration
    | function_declaration
    | dpi_import_export
    | extern_constraint_declaration
    | class_declaration
    | class_constructor_declaration
    | parameter_declaration ;
    | local_parameter_declaration
    | covergroup_declaration
    | overload_declaration
    | concurrent_assertion_item_declaration
    | ;
anonymous_program ::= program ; { anonymous_program_item } endprogram
anonymous_program_item ::=
      task_declaration
    | function_declaration
    | class_declaration
    | covergroup_declaration
    | class_constructor_declaration
    | ;
```

Syntax 19-1—Package declaration syntax (excerpt from Annex A)

The **package** declaration creates a scope that contains declarations intended to be shared among one or more compilation units, modules, macromodules, interfaces, or programs. Items within packages are generally type definitions, tasks, and functions. Items within packages cannot have hierarchical references. It is also possible to populate packages with parameters, variables, and nets. This may be useful for global items that are not conveniently passed down through the hierarchy. Variable declaration assignments within the package shall occur before any **initial**, **always**, **always_comb**, **always_latch**, or **always_ff** blocks are started, in the same way as variables declared in a compilation unit or module.

The following is an example of a package:

```
package ComplexPkg;
   typedef struct {
      float i, r;
   } Complex;

   function Complex add(Complex a, b);
      add.r = a.r + b.r;
      add.i = a.i + b.i;
```

```
        endfunction

        function Complex mul(Complex a, b);
            mul.r = (a.r * b.r) - (a.i * b.i);
            mul.i = (a.r * b.i) + (a.i * b.r);
        endfunction
    endpackage : ComplexPkg
```

19.2.1 Referencing data in packages

Packages must exist in order for the items they define to be recognized by the scopes in which they are imported.

One way to use declarations made in a package is to reference them using the class scope resolution operator `::`.

```
    ComplexPkg::Complex cout = ComplexPkg::mul(a, b);
```

An alternate method for utilizing package declarations is via the **import** statement.

data_declaration ::=	// from <u>A.2.1.3</u>
...	
| package_import_declaration	
package_import_declaration ::=	
import package_import_item { **,** package_import_item } **;**	
package_import_item ::=	
package_identifier **::** identifier	
| package_identifier **:: ***	

Syntax 19-2—Package import syntax (excerpt from <u>Annex A</u>)

The **import** statement provides direct visibility of identifiers within packages. It allows identifiers declared within packages to be visible within the current scope without a package name qualifier. Two forms of the import statement are provided: explicit import and wildcard import. Explicit import allows control over precisely which symbols are imported:

```
    import ComplexPkg::Complex;
    import ComplexPkg::add;
```

An explicit import only imports the symbols specifically referenced by the import.

In the example below, the import of the enumeration type `teeth_t` does not import the enumeration literals `ORIGINAL` and `FALSE`. In order to refer to the enumeration literal `FALSE` from package `q`, either add `import q::FALSE` or use a full package reference as in `teeth = q::FALSE;`.

```
    package p;
        typedef enum { FALSE, TRUE } bool_t;
    endpackage

    package q;
        typedef enum { ORIGINAL, FALSE } teeth_t;
    endpackage

    module top1 ;
        import p::*;
```

```
        import q::teeth_t;

        teeth_t myteeth;

        initial begin
            myteeth = q:: FALSE;  // OK:
            myteeth = FALSE;        // ERROR: Direct reference to FALSE refers to the
        end                         // FALSE enumeration literal imported from p
    endmodule

    module top2 ;
        import p::*;
        import q::teeth_t, q::ORIGINAL, q::FALSE;

        teeth_t myteeth;

        initial begin
            myteeth = FALSE;  // OK: Direct reference to FALSE refers to the
        end                    // FALSE enumeration literal imported from q
    endmodule
```

An explicit import shall be illegal if the imported identifier is declared in the same scope or explicitly imported from another package. Importing an identifier from the same package multiple times is allowed.

A wildcard import allows all identifiers declared within a package to be imported provided the identifier is not otherwise defined in the importing scope:

```
    import ComplexPkg::*;
```

A wildcard import makes each identifier within the package a candidate for import. Each such identifier is imported only when it is referenced in the importing scope and it is neither declared nor explicitly imported into the scope. Similarly, a wildcard import of an identifier is overridden by a subsequent declaration of the same identifier in the same scope. If the same identifier is wildcard imported into a scope from two different packages, the identifier shall be undefined within that scope, and an error results if the identifier is used.

19.2.2 Search order rules

Table 19-1 describes the search order rules for the declarations imported from a package. For the purposes of the discussion below, consider the following package declarations:

```
    package p;
        typedef enum { FALSE, TRUE } BOOL;
        const BOOL c = FALSE;
    endpackage

    package q;
        const int c = 0;
    endpackage
```

Table 19-1—Scoping rules for package importation

Example	Description	Scope containing a local declaration of c	Scope not containing a local declaration of c	Scope containing a declaration of c imported using import q::c	Scope containing a declaration of c imported as import q::*
`u = p::c;` `y = p::TRUE;`	A qualified package identifier is visible in any scope (without the need for an import clause).	OK Direct reference to c refers to the locally declared c. p::c refers to the c in package p.	OK Direct reference to c is illegal because it is undefined. p::c refers to the c in package p.	OK Direct reference to c refers to the c imported from q. p::c refers to the c in package p.	OK Direct reference to c refers to the c imported from q. p::c refers to the c in package p.
import p::*; . . . y = FALSE;	All declarations inside package p become potentially directly visible in the importing scope: – c – BOOL – FALSE – TRUE	OK Direct reference to c refers to the locally declared c. Direct reference to other identifiers (e.g., FALSE) refers to those implicitly imported from package p.	OK Direct reference to c refers to the c imported from package p.	OK Direct reference to c refers to the c imported from package q.	**OK** / **ERROR** c is undefined in the importing scope. Thus, a direct reference to c is illegal and results in an error. The import clause is otherwise allowed.
import p::c; . . . if(! c) ...	The imported identifiers become directly visible in the importing scope: c	ERROR "	OK Direct reference to c refers to the c imported from package p.	ERROR It shall be illegal to import an identifier defined in the importing scope.	**OK** / **ERROR** The import of p::c makes any prior reference to c illegal. Otherwise, direct reference to c refers to the c imported from package p.

When using a wildcard import, a reference to an undefined identifier that is declared within the package causes that identifier to be imported into the local scope. However, an error results if the same identifier is later declared or explicitly imported. This is shown in the following example:

```
module foo;
    import q::*;
    wire  a = c;  // This statement forces the import of q::c;
    import p::c;  // The conflict with q::c and p::c creates an error.
endmodule
```

19.3 Compilation unit support

SystemVerilog supports separate compilation using compiled units. The following terms and definitions are provided:

— **compilation unit:** A collection of one or more SystemVerilog source files compiled together.

— **compilation-unit scope:** A scope that is local to the compilation unit. It contains all declarations that lie outside of any other scope.

— **$unit:** The name used to explicitly access the identifiers in the compilation-unit scope.

The exact mechanism for defining which files constitute a compilation unit is tool-specific. However, compliant implementations shall have the same default behavior, and tools shall provide use models that allow both of the following cases:

a) All files on a given compilation command line make a single compilation unit (in which case the declarations within those files are accessible anywhere else within the constructs defined within those files).

b) Each file is a separate compilation unit (in which case the declarations in each compilation-unit scope are accessible only within its corresponding file).

The contents of files included using one or more `include directives become part of the compilation unit of the file within which they are included.

If there is a declaration that is incomplete at the end of a file, then the compilation unit including that file will extend through each successive file until there are no incomplete declarations at the end of the group of files.

The default definition of a compilation unit is defined in case b above, in which each file is a separate compilation unit.

There are other possible mappings of files to compilation units, and the mechanisms for defining them are tool-specific and may not be portable.

The compilation-unit scope can contain any item that can be defined within a package. These items are in the compilation-unit scope name space.

The following items are visible in all compilation units: modules, macromodules, primitives, programs, interfaces, and packages. Items defined in the compilation-unit scope cannot be accessed by name from outside the compilation unit. Access to the items in a compilation-unit scope can be accessed using the PLI, which must provide an iterator to traverse all the compilation units.

In Verilog, compiler directives once seen by a tool apply to all forthcoming source text. This behavior shall be supported within a separately compiled unit; however, compiler directives from one separately compiled unit shall not affect other compilation units. This may result in a difference of behavior between compiling the units separately or as a single compilation unit containing the entire source.

When an identifier is referenced within a scope, SystemVerilog follows the Verilog name search rules:

— First, the nested scope is searched (see 12.6 of IEEE Std 1364) (including nested module declarations), including any identifiers made available through package import declarations.

— Next, the compilation-unit scope is searched (including any identifiers made available through package import declarations).

— Finally, the instance hierarchy is searched (see 12.5 of IEEE Std 1364).

`$unit` is the name of the scope that encompasses a compilation unit. Its purpose is to allow the unambiguous reference to declarations at the outermost level of a compilation unit (i.e., those in the compilation-unit scope). This is done via the same class scope resolution operator used to access package items.

For example:

```
bit b;
task foo;
   int b;
   b = 5 + $unit::b;    // $unit::b is the one outside
endtask
```

The compilation-unit scope allows users to easily share declarations (e.g., types) across the unit of compilation, but without having to declare a package from which the declarations are subsequently imported. Thus, the compilation-unit scope is similar to an implicitly defined anonymous package. Because it has no name, the compilation-unit scope cannot be used with an import statement, and the identifiers declared within the scope are not accessible via hierarchical references. Within a particular compilation unit, however, the special name `$unit` can be used to explicitly access the declarations of its compilation-unit scope.

19.4 Top-level instance

The name `$root` is added to unambiguously refer to a top-level instance or to an instance path starting from the root of the instantiation tree. `$root` is the root of the instantiation tree.

For example:

```
$root.A.B      // item B within top instance A
$root.A.B.C    // item C within instance B within instance A
```

`$root` allows explicit access to the top of the instantiation tree. This is useful to disambiguate a local path (which takes precedence) from the rooted path. In Verilog, a hierarchical path is ambiguous. For example, `A.B.C` can mean the local `A.B.C` or the top-level `A.B.C` (assuming there is an instance `A` that contains an instance `B` at both the top level and in the current module). Verilog addresses that ambiguity by giving priority to the local scope and thereby preventing access to the top-level path. `$root` allows explicit access to the top level in those cases in which the name of the top-level module is insufficient to uniquely identify the path.

19.5 Module declarations

```
module_declaration ::=                                                          // from A.1.2
        module_nonansi_header [ timeunits_declaration ] { module_item }
            endmodule [ : module_identifier ]
      | module_ansi_header [ timeunits_declaration ] { non_port_module_item }
            endmodule [ : module_identifier ]
      | { attribute_instance } module_keyword [ lifetime ] module_identifier ( .* ) ;
            [ timeunits_declaration ] { module_item } endmodule [ : module_identifier ]
      | extern module_nonansi_header
      | extern module_ansi_header
module_nonansi_header ::=
        { attribute_instance } module_keyword [ lifetime ] module_identifier [ parameter_port_list ]
            list_of_ports ;
module_ansi_header ::=
        { attribute_instance } module_keyword [ lifetime ] module_identifier [ parameter_port_list ]
            [ list_of_port_declarations ] ;
module_keyword ::= module | macromodule
timeunits_declaration ::=
        timeunit time_literal ;
      | timeprecision time_literal ;
      | timeunit time_literal ;
        timeprecision time_literal ;
      | timeprecision time_literal ;
        timeunit time_literal ;
```

Syntax 19-3—Module declaration syntax (excerpt from Annex A)

In Verilog, a module must be declared apart from other modules and can only be instantiated within another module. A module declaration can appear after it is instantiated in the source text.

SystemVerilog adds the capability to nest module declarations.

```
    module m1(...); ... endmodule

    module m2(...); ... endmodule

    module m3(...);

        m1 i1(...); // instantiates the local m1 declared below
        m2 i4(...); // instantiates m2 - no local declaration
        module m1(...); ... endmodule // nested module declaration,
                                       // m1 module name is in m3's name space
    endmodule
```

19.6 Nested modules

A module can be declared within another module. The outer name space is visible to the inner module so that any name declared there can be used, unless hidden by a local name, provided the module is declared and instantiated in the same scope.

One purpose of nesting modules is to show the logical partitioning of a module without using ports. Names that are global are in the outermost scope, and names that are only used locally can be limited to local modules.

```
// This example shows a D-type flip-flop made of NAND gates
module dff_flat(input d, ck, pr, clr, output q, nq);
wire q1, nq1, q2, nq2;

    nand g1b (nq1, d, clr, q1);
    nand g1a (q1, ck, nq2, nq1);

    nand g2b (nq2, ck, clr, q2);
    nand g2a (q2, nq1, pr, nq2);

    nand g3a (q, nq2, clr, nq);
    nand g3b (nq, q1, pr, q);
endmodule


// This example shows how the flip-flop can be structured into 3 RS latches.
module dff_nested(input d, ck, pr, clr, output q, nq);
wire q1, nq1, nq2;

    module ff1;
        nand g1b (nq1, d, clr, q1);
        nand g1a (q1, ck, nq2, nq1);
    endmodule
    ff1 i1();

    module ff2;
        wire q2; // This wire can be encapsulated in ff2
        nand g2b (nq2, ck, clr, q2);
        nand g2a (q2, nq1, pr, nq2);
    endmodule
    ff2 i2();

    module ff3;
        nand g3a (q, nq2, clr, nq);
        nand g3b (nq, q1, pr, q);
    endmodule
    ff3 i3();
endmodule
```

The nested module declarations can also be used to create a library of modules that is local to part of a design.

```
module part1(....);
    module and2(input a, b, output z);
    ....
    endmodule
    module or2(input a, b, output z);
    ....
    endmodule
    ....
    and2 u1(....), u2(....), u3(....);
    .....
endmodule
```

This allows the same module name, e.g., and2, to occur in different parts of the design and represent different modules. An alternative way of handling this problem is to use configurations.

Nested modules with no ports that are not explicitly instantiated shall be implicitly instantiated once with an instance name identical to the module name. Otherwise, if they have ports and are not explicitly instantiated, they are ignored.

19.7 Extern modules

To support separate compilation, extern declarations of a module can be used to declare the ports on a module without defining the module itself. An extern module declaration consists of the keyword **extern** followed by the module name and the list of ports for the module. Both list of ports syntax (possibly with parameters) and original Verilog style port declarations can be used.

NOTE—The potential existence of defparams precludes the checking of the port connection information prior to elaboration time even for list of ports style declarations.

The following example demonstrates the usage of extern module declarations.

```
extern module m (a,b,c,d);
extern module a #(parameter size= 8, parameter type TP = logic [7:0])
                (input [size:0] a, output TP b);

module top ();
   wire [8:0] a;
   logic [7:0] b;

   m m (.*);
   a a (.*);
endmodule
```

Modules m and a are then assumed to be instantiated as follows:

```
module top ();
   m m (a,b,c,d);
   a a (a,b);
endmodule
```

If an **extern** declaration exists for a module, it is possible to use .* as the ports of the module. This usage shall be equivalent to placing the ports (and possibly parameters) of the **extern** declaration on the module.

For example:

```
extern module m (a,b,c,d);
extern module a #(parameter size = 8, parameter type TP = logic [7:0])
                (input [size:0] a, output TP b);

module m (.*);
   input a,b,c;
   output d;
endmodule

module a (.*);
   ...
endmodule
```

is equivalent to writing

```
module m (a,b,c,d);
   input a,b,c;
   output d;
endmodule

module a #(parameter size = 8, parameter type TP = logic [7:0])
         (input [size:0] a, output TP b);
   ...
endmodule
```

Extern module declarations can appear at any level of the instantiation hierarchy, but are visible only within the level of hierarchy in which they are declared. An extern module declaration shall match the actual module declaration's port and parameter lists in correspondence of names, positions, and their equivalent types.

19.8 Port declarations

```
inout_declaration ::=                                                           // from A.2.1.2
        inout net_port_type list_of_port_identifiers
input_declaration ::=
        input net_port_type list_of_port_identifiers
      | input variable_port_type list_of_variable_identifiers
output_declaration ::=
        output net_port_type list_of_port_identifiers
      | output variable_port_type  list_of_variable_port_identifiers
interface_port_declaration ::=
        interface_identifier list_of_interface_identifiers
      | interface_identifier . modport_identifier  list_of_interface_identifiers
ref_declaration ::= ref variable_port_type list_of_port_identifiers
net_port_type[33] ::=                                                           // from A.2.2.1
        [ net_type ] data_type_or_implicit
variable_port_type ::= var_data_type
var_data_type ::= data_type | var data_type_or_implicit
```

33. When a net_port_type contains a data_type, it shall only be legal to omit the explicit net_type when declaring an **inout** port.

Syntax 19-4—Port declaration syntax (excerpt from Annex A)

With SystemVerilog, a port can be a declaration of an interface, an event, or a variable or net of any allowed data type, including an array, a structure, or a union. Within this subclause, the term *port kind* is used to mean any of the net type keywords, or the keyword **var**, which are used to explicitly declare a port of one of these kinds. If these keywords are omitted in a port declaration, there are default rules for determining the port kind.

```
typedef struct {
            bit isfloat;
            union { int i; shortreal f; } n;
} tagged_st; // named structure
```

```
    module mh1 (input int in1, input shortreal in2, output tagged_st out);
       ...
    endmodule
```

For the first port, if neither a type nor a direction is specified, then it shall be assumed to be a member of a port list, and any port direction or type declarations must be declared after the port list. This is compatible with the Verilog-1995 syntax. If the first port kind or data type is specified, but no direction is specified, then the port direction shall default to **inout**. If the first port direction is specified, but no port kind or data type is specified, then the port shall default to a net of net type **wire**. This default net type can be changed using the `default_nettype` compiler directive, as in Verilog.

```
    // Any declarations must follow the port list because first port does not
    // have either a direction or type specified; Port directions default to
    inout
    module mh4(x, y);
       wire x;
       tri0 y;
       ...
    endmodule
```

For subsequent ports in the port list, if the direction and the port kind and data type are omitted, then the direction and any port kind and data type are inherited from the previous port. If the direction is omitted, but a port kind or data type is present, then the direction is inherited from the previous port. If the direction is present, but the port kind and data type are omitted, then the port shall default to a net of net type **wire**. This default net type can be changed using the `default_nettype` compiler directive, as in Verilog.

```
    // second port inherits its direction and data type from previous port
    module mh3 (input byte a, b);
       ...
    endmodule
```

For **input** and **inout** ports, if the port kind is omitted, then the port shall default to a net of net type **wire**. This default net type can be changed using the `default_nettype` compiler directive, as in Verilog.

```
    // the inout port defaults to a net of net type wire
    module mh2 (inout integer a);
    ...
    endmodule
```

For **output** ports, if the port kind is omitted, the default port kind depends on how the data type is specified. If the port is declared without the data_type syntax, then the port kind defaults to a net of the default net type. This provides backward compatibility with Verilog. If the data type is declared with the data_type syntax, the port kind defaults to variable.

Generic interface ports cannot be declared using the Verilog-1995 list of ports style. Generic interface ports can only be declared by using a list of port declaration style.

```
    module cpuMod(interface d, interface j);
       ...
    endmodule
```

19.9 List of port expressions

IEEE Std 1364-2001 created a *list_of_port_declarations* alternate style that minimized the duplication of data used to specify the ports of a module. SystemVerilog adds an explicitly named port declaration to that

style, allowing elements of arrays and structures, concatenations of elements, and assignment pattern expressions of elements declared in a module, interface, or program to be specified on the port list.

Like explicitly named ports in a module port declaration, port identifiers exist in their own name space for each port list. When a port item is just a simple port identifier, that identifier is used as both a reference to an interface item and a port identifier. Once a port identifier has been defined, there shall not be another port definition with this same name.

For example:

```
module mymod (
    output .P1(r[3:0]),
    output .P2(r[7:4]),
    ref    .Y(x),
    input bit R );

    logic [7:0] r;
    int x;
    ...
endmodule
```

The self-determined type of the port expression becomes the type for the port. The port expression shall not be considered an assignment-like context. The port expression must resolve to a legal expression for type of module port (see 19.12). The port expression is optional because ports can be defined that do not connect to anything internal to the port.

19.10 Time unit and precision

SystemVerilog has a time unit and precision declaration that has the equivalent functionality of the `timescale` compiler directives in Verilog. Use of these declarations removes the file order dependency problems with compiler directives. The time unit and precision can be declared by the **timeunit** and **timeprecision** keywords, respectively, and set to a time literal, which must be a power of 10 units. For example:

```
timeunit 100ps;
timeprecision 10fs;
```

There shall be at most one time unit and one time precision for any module, program, package, or interface definition or in any compilation-unit scope. This shall define a time scope. If specified, the **timeunit** and **timeprecision** declarations shall precede any other items in the current time scope. The **timeunit** and **timeprecision** declarations can be repeated as later items, but must match the previous declaration within the current time scope.

If a **timeunit** is not specified in the module, program, package, or interface definition, then the time unit shall be determined using the following rules of precedence:

a)   If the module or interface definition is nested, then the time unit shall be inherited from the enclosing module or interface (programs and packages cannot be nested).

b)   Else, if a `timescale` directive has been previously specified (within the compilation unit), then the time unit shall be set to the units of the last `timescale` directive.

c)   Else, if the compilation-unit scope specifies a time unit (outside all other declarations), then the time unit shall be set to the time units of the compilation unit.

d)   Else, the default time unit shall be used.

The time unit of the compilation-unit scope can only be set by a **timeunit** declaration, not a `timescale directive. If it is not specified, then the default time unit shall be used.

If a **timeprecision** is not specified in the current time scope, then the time precision shall be determined following the same precedence as with time units.

The global time precision is the minimum of all the time precision statements and the smallest time precision argument of all the `timescale compiler directives (known as the precision of the time unit of the simulation in 19.8 of IEEE Std 1364) in the design. The **step** time unit is equal to the global time precision.

19.11 Module instances

```
module_instantiation ::=                                                    // from A.4.1.1
        module_identifier [ parameter_value_assignment ] hierarchical_instance { , hierarchical_instance };
parameter_value_assignment ::= # ( [ list_of_parameter_assignments ] )
list_of_parameter_assignments ::=
        ordered_parameter_assignment { , ordered_parameter_assignment }
      | named_parameter_assignment { , named_parameter_assignment }
ordered_parameter_assignment ::= param_expression
named_parameter_assignment ::= . parameter_identifier ( [ param_expression ] )
hierarchical_instance ::= name_of_instance ( [ list_of_port_connections ] )
name_of_instance ::= instance_identifier { unpacked_dimension }
list_of_port_connections[16] ::=
        ordered_port_connection { , ordered_port_connection }
      | named_port_connection { , named_port_connection }
ordered_port_connection ::= { attribute_instance } [ expression ]
named_port_connection ::=
        { attribute_instance } . port_identifier [ ( [ expression ] ) ]
      | { attribute_instance } .*
param_expression ::= mintypmax_expression | data_type                        // from A.8.3
```

Syntax 19-5—Module instance syntax (excerpt from Annex A)

A module can be used (instantiated) in two ways, hierarchical or top level. Hierarchical instantiation allows more than one instance of the same type. The module name can be a module previously declared or one declared later. Actual parameters can be named or ordered. Port connections can be named, ordered, or implicitly connected. They can be nets, variables, or other kinds of interfaces, events, or expressions. See 19.12 for the connection rules.

Consider an ALU accumulator (`alu_accum`) example module that includes instantiations of an ALU module, an accumulator register (`accum`) module, and a sign-extension (`xtend`) module. The module headers for the three instantiated modules are shown in the following example code:

```
    module alu (
       output reg [7:0] alu_out,
       output reg zero,
       input [7:0] ain, bin,
       input [2:0] opcode);
       // RTL code for the alu module
    endmodule
```

```
module accum (
    output reg [7:0] dataout,
    input [7:0] datain,
    input clk, rst_n);
    // RTL code for the accumulator module
endmodule

module xtend (
    output reg [7:0] dout,
    input din,
    input clk, rst_n);
    // RTL code for the sign-extension module
endmodule
```

19.11.1 Instantiation using positional port connections

Verilog has always permitted instantiation of modules using positional port connections, as shown in the `alu_accum1` module example below.

```
module alu_accum1 (
    output [15:0] dataout,
    input [7:0] ain, bin,
    input [2:0] opcode,
    input clk, rst_n);
    wire [7:0] alu_out;

    alu alu (alu_out, , ain, bin, opcode);
    accum accum (dataout[7:0], alu_out, clk, rst_n);
    xtend xtend (dataout[15:8], alu_out[7], clk, rst_n);
endmodule
```

As long as the connecting variables are ordered correctly and are the same size as the instance ports to which they are connected, there shall be no warnings and the simulation shall work as expected.

19.11.2 Instantiation using named port connections

Verilog has always permitted instantiation of modules using named port connections, as shown in the `alu_accum2` module example below.

```
module alu_accum2 (
    output [15:0] dataout,
    input [7:0] ain, bin,
    input [2:0] opcode,
    input clk, rst_n);
    wire [7:0] alu_out;

    alu   alu   (.alu_out(alu_out), .zero(),
                 .ain(ain), .bin(bin), .opcode(opcode));
    accum accum (.dataout(dataout[7:0]), .datain(alu_out),
                 .clk(clk), .rst_n(rst_n));
    xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]),
                 .clk(clk), .rst_n(rst_n));
endmodule
```

Named port connections do not have to be ordered the same as the ports of the instantiated module. The variables connected to the instance ports must be the same size, or a port-size mismatch warning shall be reported.

19.11.3 Instantiation using implicit .name port connections

SystemVerilog adds the capability to implicitly instantiate ports using a `.name` syntax if the instance port name and equivalent type match the connecting declaration port name and type. This enhancement eliminates the requirement to list a port name twice when the port name and declaration name are the same, while still listing all of the ports of the instantiated module for documentation purposes.

In the following `alu_accum3` example, all of the ports of the instantiated `alu` module match the names of the declarations connected to the ports, except for the unconnected zero port, which is listed using a named port connection, showing that the port is unconnected. Implicit `.name` port connections are made for all name and equivalent type matching connections on the instantiated module.

In the same `alu_accum3` example, the `accum` module has an 8-bit port called `dataout` that is connected to a 16-bit bus called `dataout`. Because the internal and external sizes of `dataout` do not match, the port must be connected by name, showing which bits of the 16-bit bus are connected to the 8-bit port. The `datain` port on the `accum` is connected to a bus by a different name (`alu_out`); therefore, this port is also connected by name. The `clk` and `rst_n` ports are connected using implicit `.name` port connections. Also in the same `alu_accum3` example, the `xtend` module has an 8-bit output port called `dout` and a 1-bit input port called `din`. Because neither of these port names matches the names (or sizes) of the connecting declarations, both are connected by name. The `clk` and `rst_n` ports are connected using implicit `.name` port connections.

```
module alu_accum3 (
    output [15:0] dataout,
    input [7:0] ain, bin,
    input [2:0] opcode,
    input clk, rst_n);
    wire [7:0] alu_out;

    alu   alu   (.alu_out, .zero(), .ain, .bin, .opcode);
    accum accum (.dataout(dataout[7:0]), .datain(alu_out), .clk, .rst_n);
    xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]), .clk, .rst_n);
endmodule
```

A *.port_identifier* port connection is semantically equivalent to the named port connection *.port_identifier*(`port_identifier`) with the following exceptions:

— The port connection shall not create an implicit wire declaration.

— The declarations on each side of the port connection shall have equivalent data types.

— An implicit *.port_identifier* port connection between nets of two dissimilar net types shall generate an error when it is a warning in an explicit named port connection as required by IEEE Std 1364.

19.11.4 Instantiation using implicit .* port connections

SystemVerilog adds the capability to implicitly instantiate ports using a `.*` syntax for all ports where the instance port name matches the connecting port name and their data types are equivalent. This enhancement eliminates the requirement to list any port where the name and type of the connecting declaration match the name and equivalent type of the instance port. This implicit port connection style is used to indicate that all port names and types match the connections where emphasis is placed only on the exception ports. The implicit `.*` port connection syntax can greatly facilitate rapid block-level testbench generation where all of the testbench declarations are chosen to match the instantiated module port names and types.

In the following `alu_accum4` example, all of the ports of the instantiated `alu` module match the names of the variables connected to the ports, except for the unconnected zero port, which is listed using a named port connection, showing that the port is unconnected. The implicit `.*` port connection syntax connects all other ports on the instantiated module.

In the same `alu_accum4` example, the `accum` module has an 8-bit port called `dataout` that is connected to a 16-bit bus called `dataout`. Because the internal and external sizes of `dataout` do not match, the port must be connected by name, showing which bits of the 16-bit bus are connected to the 8-bit port. The `datain` port on the `accum` is connected to a bus by a different name (`alu_out`); therefore, this port is also connected by name. The `clk` and `rst_n` ports are connected using implicit `.*` port connections. Also in the same `alu_accum4` example, the `xtend` module has an 8-bit output port called `dout` and a 1-bit input port called `din`. Because neither of these port names matches the names (or sizes) of the connecting declarations, both are connected by name. The `clk` and `rst_n` ports are connected using implicit `.*` port connections.

```
module alu_accum4 (
    output [15:0] dataout,
    input [7:0] ain, bin,
    input [2:0] opcode,
    input clk, rst_n);
    wire [7:0] alu_out;

    alu    alu   (.*, .zero());
    accum accum (.*, .dataout(dataout[7:0]), .datain(alu_out));
    xtend xtend (.*, .dout(dataout[15:8]), .din(alu_out[7]));
endmodule
```

An implicit `.*` port connection is semantically equivalent to a default `.name` port connection for every port declared in the instantiated module with the exception that `.*` does not create a sufficient reference for a wildcard import of a name from a package. A named port connection can be mixed with a `.*` connection to override a port connection to a different expression or to leave a port unconnected. A named or implicit `.name` connection can be mixed with a `.*` connection to create a sufficient reference for a wildcard import of a name from a package.

When the implicit `.*` port connection is mixed in the same instantiation with named port connections, the implicit `.*` port connection token can be placed anywhere in the port list. The `.*` token can only appear at most once in the port list.

Modules can be instantiated into the same parent module using any combination of legal positional, named, implicit `.name` connected and implicit `.*` connected instances, as shown in `alu_accum5` example below.

```
module alu_accum5 (
    output [15:0] dataout,
    input [7:0] ain, bin,
    input [2:0] opcode,
    input clk, rst_n);
    wire [7:0] alu_out;

    // mixture of named port connections and
    // implicit .name port connections
    alu    alu   (.ain(ain), .bin(bin), .alu_out, .zero(), .opcode);

    // positional port connections
    accum  accum  (dataout[7:0], alu_out, clk, rst_n);

    // mixture of named port connections and implicit .* port connections
    xtend  xtend  (.dout(dataout[15:8]), .*, .din(alu_out[7]));
```

```
    endmodule
```

19.12 Port connection rules

SystemVerilog extends Verilog port connections by making values of all data types on variables and nets available to passthrough ports. It does this by allowing both sides of a port connection to have assignment-compatible data types and by allowing continuous assignments to variables. It also creates a new type of port qualifier, **ref**, to allow shared variable behavior across a port by passing a hierarchical reference.

19.12.1 Port connection rules for variables

If a port declaration has a variable data type, then its direction controls how it can be connected when instantiated, as follows:

— An **input** port can be connected to any expression of a compatible data type. A continuous assignment shall be implied when a variable is connected to an input port declaration. Assignments to variables declared as an input port shall be illegal. If left unconnected, the port shall have the default initial value corresponding to the data type.

— An **output** port can be connected to a variable (or a concatenation) of a compatible data type. A continuous assignment shall be implied when a variable is connected the output port of an instance. Procedural or continuous assignments to a variable connected to the output port of an instance shall be illegal.

— An **output** port can be connected to a net (or a concatenation) of a compatible data type. In this case, multiple drivers shall be permitted on the net as in Verilog.

— A variable data type is not permitted on either side of an **inout** port.

— A **ref** port shall be connected to an equivalent variable data type. References to the port variable shall be treated as hierarchical references to the variable to which it is connected in its instantiation. This kind of port cannot be left unconnected. See 6.9.2.

19.12.2 Port connection rules for nets

If a port declaration has a net type, such as **wire**, then its direction controls how it can be connected as follows:

— An **input** can be connected to any expression of a compatible data type. If left unconnected, it shall have the value 'z.

— An **output** can be connected to a net or variable (or a concatenation of nets or variables) of a compatible data type.

— An **inout** can be connected to a net (or a concatenation of nets) of a compatible data type or left unconnected, but cannot be connected to a variable.

If there is a data type difference between the port declaration and connection, an initial value change event can be caused at time zero.

19.12.3 Port connection rules for interfaces

A port declaration can be a generic interface or named interface type. An interface port instance must always be connected to an interface instance or a higher level interface port. An interface port cannot be left unconnected.

If a port declaration has a generic interface type, then it can be connected to an interface instance of any type. If a port declaration has a named interface type, then it must be connected to an interface instance of the identical type.

19.12.4 Compatible port types

The same rules are used for compatible port types as for assignment compatibility. SystemVerilog does not change any of the other port connection compatibility rules.

19.12.5 Unpacked array ports and arrays of instances

For an unpacked array port, the port and the array connected to the port must have the same number of unpacked dimensions, and each dimension of the port must have the same size as the corresponding dimension of the array being connected.

If the size and type of the port connection match the size and type of a single instance port, the connection shall be made to each instance in an array of instances.

If the port connection is an unpacked array, the slowest varying unpacked array dimensions of each port connection shall be compared with the dimensions of the instance array. If they match exactly in size, each element of the port connection shall be matched to the port left index to left index, right index to right index. If they do not match it shall be considered an error.

For example:

```
module child(output o, input i[5]);
    //...
endmodule : child

module parent(output o[8][4],
              input  i[8][4][5] );
    child c[8][4](o,i);
    //...
endmodule : parent
```

If the port connection is a packed array, each instance shall get a part-select of the port connection, starting with all right-hand indices to match the rightmost part-select and iterating through the rightmost dimension first. Too many or too few bits to connect all the instances shall be considered an error.

In the example below, a two-dimensional array of DFF instances is connected to form *M* pipelines with *N* stages.

```
module MxN_pipeline #(M=3,N=4)
    (input  [M-1:0] in, output [M-1:0] out, input clk);

    typedef logic T [M-1:0][1:N];
    T Ins, Outs;

    DFF dff[M-1:0][1:N](Outs, Ins, clk);

    for (genvar I = M-1; I >= 0; I--) begin
       for (genvar J = 1; J <= N; J++) begin


          case (J)
                  1: begin
                         assign out[I] = Outs[I][1];
                         assign Ins[I][J] = Outs[I][2];
                     end
              default: assign Ins[I][J] = Outs[I][J+1];
```

```
                        N: assign Ins[I][N] = in[I];
            endcase
        end
    end
  endmodule : MxN_pipeline
```

19.13 Name spaces

SystemVerilog has eight name spaces for identifiers: two are global (definitions name space and package name space), two are global to the compilation unit (compilation unit name space and text macro name space), and four are local. The eight name spaces are described as follows:

a) The *definitions name space* unifies all the non-nested **module**, **macromodule**, **primitive**, **program**, and **interface** identifiers defined outside of all other declarations. Once a name is used to define a module, macromodule, primitive, program, or interface within one compilation unit, the name shall not be used again (in any compilation unit) to declare another non-nested module, macromodule, primitive, program, or interface outside of all other declarations. This is compatible with the *definitions name space* as defined in Verilog.

b) The *package name space* unifies all the **package** identifiers defined among all compilation units. Once a name is used to define a package within one compilation unit, the name shall not be used again to declare another package within any compilation unit.

c) The *compilation-unit scope name space* exists outside the **module**, **macromodule**, **interface**, **package**, **program**, and **primitive** constructs. It unifies the definitions of the functions, tasks, parameters, named events, net declarations, variable declarations, and user-defined types within the compilation-unit scope.

d) The *text macro name space* is global within the compilation unit. Because text macro names are introduced and used with a leading ` character, they remain unambiguous with any other name space. The text macro names are defined in the linear order of appearance in the set of input files that make up the compilation unit. Subsequent definitions of the same name override the previous definitions for the balance of the input files.

e) The *module name space* is introduced by the **module**, **macromodule**, **interface**, **package**, **program**, and **primitive** constructs. It unifies the definition of modules, macromodules, interfaces, programs, functions, tasks, named blocks, instance names, parameters, named events, net declarations, variable declarations, and user-defined types within the enclosing construct.

f) The *block name space* is introduced by named or unnamed blocks, the **specify**, **function**, and **task** constructs. It unifies the definitions of the named blocks, functions, tasks, parameters, named events, variable type of declaration, and user-defined types within the enclosing construct.

g) The *port name space* is introduced by the **module**, **macromodule**, **interface**, **primitive**, and **program** constructs. It provides a means of structurally defining connections between two objects that are in two different name spaces. The connection can be unidirectional (either **input** or **output**) or bidirectional (**inout** or **ref**). The port name space overlaps the module and the block name spaces. Essentially, the port name space specifies the type of connection between names in different name spaces. The port type of declarations includes **input**, **output**, **inout**, and **ref**. A port name introduced in the port name space can be reintroduced in the module name space by declaring a variable or a net with the same name as the port name.

h) The *attribute name space* is enclosed by the (* and *) constructs attached to a language element (see 3.8). An attribute name can be defined and used only in the attribute name space. Any other type of name cannot be defined in this name space.

19.14 Hierarchical names

Hierarchical names are also called *nested identifiers*. They consist of instance names separated by periods, where an instance name can be an array element. The instance name $root refers to the top of the instantiated design and is used to unambiguously gain access to the top of the design.

```
$root.mymodule.u1 // absolute name
u1.struct1.field1 // u1 must be visible locally or above, including globally
adder1[5].sum
```

Nested identifiers can be read (in expressions), written (in assignments or in task or function calls) or triggered off (in event expressions). They can also be used as task or function names.

20. Interfaces

20.1 Introduction

NOTE—The communication between blocks of a digital system is a critical area that can affect everything from ease of RTL coding to hardware-software partitioning to performance analysis to bus implementation choices and protocol checking. The interface construct in SystemVerilog was specifically created to encapsulate the communication between block, allowing a smooth migration from abstract system-level design through successive refinement down to lower level register-transfer and structural views of the design. By encapsulating the communication between blocks, the interface construct also facilitates design reuse. The inclusion of interface capabilities is one of the major advantages of SystemVerilog.

At its lowest level, an interface is a named bundle of nets or variables. The interface is instantiated in a design and can be accessed through a port as a single item, and the component nets or variables referenced where needed. A significant proportion of a Verilog design often consists of port lists and port connection lists, which are just repetitions of names. The ability to replace a group of names by a single name can significantly reduce the size of a description and improve its maintainability.

Additional power of the interface comes from its ability to encapsulate functionality as well as connectivity, making an interface, at its highest level, more like a class template. An interface can have parameters, constants, variables, functions, and tasks. The types of elements in an interface can be declared, or the types can be passed in as parameters. The member variables and functions are referenced relative to the instance name of the interface as instance member. Thus, modules that are connected via an interface can simply call the task or function members of that interface to drive the communication. With the functionality thus encapsulated in the interface and isolated from the module, the abstraction level and/or granularity of the communication protocol can be easily changed by replacing the interface with a different interface containing the same members, but implemented at a different level of abstraction. The modules connected via the interface do not need to change at all.

To provide direction information for module ports and to control the use of tasks and functions within particular modules, the **modport** construct is provided. As the name indicates, the directions are those seen from the module.

In addition to task and function methods, an interface can also contain processes (i.e., **initial** or **always** blocks) and continuous assignments, which are useful for system-level modeling and testbench applications. This allows the interface to include, for example, its own protocol checker that automatically verifies that all modules connected via the interface conform to the specified protocol. Other applications, such as functional coverage recording and reporting, protocol checking, and assertions can also be built into the interface.

The methods can be abstract, i.e., defined in one module and called in another, using the export and import constructs. This could be coded using hierarchical path names, but this would impede reuse because the names would be design-specific. A better way is to declare the task and function names in the interface and to use local hierarchical names from the interface instance for both definition and call. Broadcast communication is modeled by **forkjoin** tasks, which can be defined in more than one module and executed concurrently.

20.2 Interface syntax

```
interface_declaration ::=                                                                    // from A.1.2
        interface_nonansi_header [ timeunits_declaration ] { interface_item }
            endinterface [ : interface_identifier ]
      | interface_ansi_header [ timeunits_declaration ] { non_port_interface_item }
            endinterface [ : interface_identifier ]
      | { attribute_instance } interface interface_identifier ( .* ) ;
            [ timeunits_declaration ] { interface_item }
        endinterface [ : interface_identifier ]
      | extern interface_nonansi_header
      | extern interface_ansi_header
interface_nonansi_header ::=
        { attribute_instance } interface [ lifetime ] interface_identifier
            [ parameter_port_list ] list_of_ports ;
interface_ansi_header ::=
        {attribute_instance } interface [ lifetime ] interface_identifier
            [ parameter_port_list ] [ list_of_port_declarations ] ;
modport_declaration ::= modport modport_item { , modport_item } ;                             // from A.2.9
modport_item ::= modport_identifier ( modport_ports_declaration { , modport_ports_declaration } )
modport_ports_declaration ::=
        { attribute_instance } modport_simple_ports_declaration
      | { attribute_instance } modport_tf_ports_declaration
      | { attribute_instance } modport_clocking_declaration
modport_clocking_declaration ::= clocking clocking_identifier
modport_simple_ports_declaration ::=
        port_direction modport_simple_port { , modport_simple_port }
modport_simple_port ::=
        port_identifier
      | . port_identifier ( [ expression ] )
modport_tf_ports_declaration ::=
        import_export modport_tf_port { , modport_tf_port }
modport_tf_port ::=
        method_prototype
      | tf_identifier
import_export ::= import | export
interface_instantiation ::=                                                                  // from A.4.1.2
        interface_identifier [ parameter_value_assignment ]
            hierarchical_instance { , hierarchical_instance } ;
```

Syntax 20-1—Interface syntax (excerpt from Annex A)

The interface construct provides a new hierarchical structure. It can contain smaller interfaces and can be passed through ports.

The aim of interfaces is to encapsulate communication. At the lower level, this means bundling variables and nets in interfaces and can impose access restrictions with port directions in modports. The modules can be made generic so that the interfaces can be changed. The following examples show these features. At a higher level of abstraction, communication can be done by tasks and functions. Interfaces can include task

and function definitions or just task and function prototypes (see 20.6.1) with the definition in one module (server/slave) and the call in another (client/master).

A simple interface declaration is as follows (see Syntax 20-1 for the complete syntax):

```
interface identifier;
    ...
    interface_items
    ...
endinterface [ : identifier ]
```

An interface can be instantiated hierarchically like a module, with or without ports. For example:

```
myinterface #(100) scalar1(), vector[9:0]();
```

In this example, 11 instances of the interface of type `myinterface` have been instantiated, and the first parameter within each interface is changed to 100. One `myinterface` instance is instantiated with the name `scalar1`, and an array of 10 `myinterface` interfaces are instantiated with instance names `vector[9]` to `vector[0]`.

Interfaces can be declared and instantiated in modules (either flat or hierarchical), but modules can neither be declared nor instantiated in interfaces.

Verilog does not permit a **defparam** statement within an array of instances to modify a parameter outside the hierarchy of the instance defining the **defparam** (see 12.2.1 of IEEE Std 1364). In SystemVerilog, a **defparam** within an instance whose port actuals refer to an arrayed interface shall be subject to the same restriction: a **defparam** shall not modify a parameter outside the hierarchy of such an instance.

The simplest use of an interface is to bundle wires, as illustrated in the examples below.

If the actual of an interface port connection is a hierarchical reference to an interface or a modport of a hierarchically referenced interface, the hierarchical reference shall refer to an interface instance and shall not resolve through an arrayed instance or a **generate** block.

20.2.1 Example without using interfaces

This example shows a simple bus implemented without interfaces. The logic type, as used in this example, can replace wire and reg if no resolution of multiple drivers is needed.

```
module memMod( input    bit req,
                        bit clk,
                        bit start,
                        logic [1:0] mode,
                        logic [7:0] addr,
               inout    wire [7:0] data,
               output   bit gnt,
                        bit rdy );
    logic avail;


    ...
endmodule

module cpuMod(
    input    bit clk,
             bit gnt,
             bit rdy,
```

```
        inout    wire [7:0] data,
        output   bit req,
                 bit start,
                 logic [7:0] addr,
                 logic [1:0] mode );
        ...
    endmodule


    module top;
        logic req, gnt, start, rdy; // req is logic not bit here
        logic clk = 0;
        logic [1:0] mode;
        logic [7:0] addr;
        wire  [7:0] data;

        memMod mem(req, clk, start, mode, addr, data, gnt, rdy);
        cpuMod cpu(clk, gnt, rdy, data, req, start, addr, mode);

    endmodule
```

20.2.2 Interface example using a named bundle

The simplest form of a SystemVerilog interface is a bundled collection of variables or nets. When an interface is referenced as a port, the variables and nets in it are assumed to have **ref** and **inout** access, respectively. The following interface example shows the basic syntax for defining, instantiating, and connecting an interface. Usage of the SystemVerilog interface capability can significantly reduce the amount of code required to model port connections.

```
    interface simple_bus; // Define the interface
        logic req, gnt;
        logic [7:0] addr, data;
        logic [1:0] mode;
        logic start, rdy;
    endinterface: simple_bus

    module memMod(simple_bus a, // Access the simple_bus interface
                  input bit clk);
        logic avail;
        // When memMod is instantiated in module top, a.req is the req
        // signal in the sb_intf instance of the 'simple_bus' interface
        always @(posedge clk) a.gnt <= a.req & avail;
    endmodule

    module cpuMod(simple_bus b, input bit clk);
        ...
    endmodule

    module top;
        logic clk = 0;

        simple_bus sb_intf(); // Instantiate the interface

        memMod mem(sb_intf, clk); // Connect the interface to the module instance
        cpuMod cpu(.b(sb_intf), .clk(clk)); // Either by position or by name

    endmodule
```

In the preceding example, if the same identifier, `sb_intf`, had been used to name the `simple_bus` interface in the `memMod` and `cpuMod` module headers, then implicit port declarations also could have been used to instantiate the `memMod` and `cpuMod` modules into the top module, as shown below.

```
module memMod (simple_bus sb_intf, input bit clk);
    ...
endmodule

module cpuMod (simple_bus sb_intf, input bit clk);
    ...
endmodule

module top;
    logic clk = 0;

    simple_bus sb_intf();

    memMod mem (.*);  // implicit port connections
    cpuMod cpu (.*);  // implicit port connections

endmodule
```

20.2.3 Interface example using a generic bundle

A module header can be created with an unspecified interface reference as a placeholder for an interface to be selected when the module itself is instantiated. The unspecified interface is referred to as a *generic interface reference*.

This generic interface reference can only be declared by using the list of port declaration style of reference. It shall be illegal to declare such a generic interface reference using the Verilog-1995 list of ports style.

The following interface example shows how to specify a generic interface reference in a module definition:

```
// memMod and cpuMod can use any interface
module memMod (interface a, input bit clk);
    ...
endmodule

module cpuMod(interface b, input bit clk);
    ...
endmodule

interface simple_bus; // Define the interface
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
    logic start, rdy;
endinterface: simple_bus

module top;
    logic clk = 0;

    simple_bus sb_intf(); // Instantiate the interface

    // Reference the sb_intf instance of the simple_bus
    // interface from the generic interfaces of the
    // memMod and cpuMod modules
```

```
        memMod mem (.a(sb_intf), .clk(clk));
        cpuMod cpu (.b(sb_intf), .clk(clk));

    endmodule
```

An implicit port cannot be used to reference a generic interface. A named port must be used to reference a generic interface, as shown below.

```
    module memMod (interface a, input bit clk);
        ...
    endmodule

    module cpuMod (interface b, input bit clk);
        ...
    endmodule

    module top;
        logic clk = 0;

        simple_bus sb_intf();

        memMod mem (.*, .a(sb_intf)); // partial implicit port connections
        cpuMod cpu (.*, .b(sb_intf)); // partial implicit port connections

    endmodule
```

20.3 Ports in interfaces

One limitation of simple interfaces is that the nets and variables declared within the interface are only used to connect to a port with the same nets and variables. To share an external net or variable, one that makes a connection from outside of the interface as well as forming a common connection to all module ports that instantiate the interface, an interface port declaration is required. The difference between nets or variables in the interface port list and other nets or variables within the interface is that only those in the port list can be connected externally by name or position when the interface is instantiated.

```
    interface i1 (input a, output b, inout c);
        wire d;
    endinterface
```

The wires a, b, and c can be individually connected to the interface and thus shared with other interfaces.

The following example shows how to specify an interface with inputs, allowing a wire to be shared between two instances of the interface:

```
    interface simple_bus (input bit clk); // Define the interface
        logic req, gnt;
        logic [7:0] addr, data;
        logic [1:0] mode;
        logic start, rdy;
    endinterface: simple_bus

    module memMod(simple_bus a); // Uses just the interface
        logic avail;

        always @(posedge a.clk) // the clk signal from the interface
            a.gnt <= a.req & avail; // a.req is in the 'simple_bus' interface
```

```
       endmodule

    module cpuMod(simple_bus b);
          ...
       endmodule

    module top;
       logic clk = 0;

       simple_bus sb_intf1(clk); // Instantiate the interface
       simple_bus sb_intf2(clk); // Instantiate the interface

       memMod mem1(.a(sb_intf1)); // Reference simple_bus 1 to memory 1
       cpuMod cpu1(.b(sb_intf1));
       memMod mem2(.a(sb_intf2)); // Reference simple_bus 2 to memory 2
       cpuMod cpu2(.b(sb_intf2));

       endmodule
```

In the preceding example, the instantiated interface names do not match the interface names used in the memMod and cpuMod modules; therefore, implicit port connections cannot be used for this example.

20.4 Modports

To restrict interface access within a module, there are **modport** lists with directions declared within the interface. The keyword **modport** indicates that the directions are declared as if inside the module.

```
    interface i2;
       wire a, b, c, d;
       modport master (input a, b, output c, d);
       modport slave (output a, b, input c, d);
    endinterface
```

In this example, the **modport** list name (master or slave) can be specified in the module header, where the interface name selects an interface and the **modport** name selects the appropriate directional information for the interface signals accessed in the module header.

```
    module m (i2.master i);
          ...
    endmodule

    module s (i2.slave i);
          ...
    endmodule

    module top;
       i2 i();

       m u1(.i(i));
       s u2(.i(i));
    endmodule
```

The syntax of interface_name.modport_name reference_name gives a local name for a hierarchical reference. This technique can be generalized to any interface with a given modport name by writing **interface**.modport_name reference_name.

The **modport** list name (master or slave) can also be specified in the port connection with the module instance, where the **modport** name is hierarchical from the interface instance.

```
module m (i2 i);
    ...
endmodule

module s (i2 i);
    ...
endmodule

module top;
    i2 i();

    m u1(.i(i.master));
    s u2(.i(i.slave));
endmodule
```

If a port connection specifies a **modport** list name in both the module instance and module header declaration, then the two **modport** list names shall be identical.

All of the names used in a **modport** declaration shall be declared by the same interface as the modport itself. In particular, the names used shall not be those declared by another enclosing interface, and a modport declaration shall not implicitly declare new ports.

The following interface declarations would be illegal:

```
interface i;
    wire x, y;

    interface illegal_i;
        wire a, b, c, d;
        // x, y not declared by this interface
        modport master(input a, b, x, output c, d, y);
        modport slave(input a, b, x, output c, d, y);
    endinterface : illegal_i

endinterface : i

interface illegal_i;
    // a, b, c, d not declared by this interface
    modport master(input a, b, output c, d);
    modport slave(output a, b, output c, d);
endinterface : illegal_i
```

Adding modports to an interface does not require that any of the modports be used when the interface is used. If no **modport** is specified in the module header or in the port connection, then all the nets and variables in the interface are accessible with direction **inout** or **ref**, as in the examples above.

20.4.1 Example of named port bundle

This interface example shows how to use modports to control signal directions as in port declarations. It uses the modport name in the module definition.

```
interface simple_bus (input bit clk); // Define the interface
    logic req, gnt;
    logic [7:0] addr, data;
```

```
    logic [1:0] mode;
    logic start, rdy;

    modport slave (input req, addr, mode, start, clk,
                   output gnt, rdy,
                   ref data);

    modport master(input gnt, rdy, clk,
                   output req, addr, mode, start,
                   ref data);

endinterface: simple_bus

module memMod (simple_bus.slave a); // interface name and modport name
    logic avail;

    always @(posedge a.clk) // the clk signal from the interface
        a.gnt <= a.req & avail; // the gnt and req signal in the interface
endmodule

module cpuMod (simple_bus.master b);
    ...
endmodule

module top;
    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate the interface

    initial repeat(10) #10 clk++;

    memMod mem(.a(sb_intf)); // Connect the interface to the module instance
    cpuMod cpu(.b(sb_intf));
endmodule
```

20.4.2 Example of connecting port bundle

This interface example shows how to use modports to restrict interface signal access and control their direction. It uses the modport name in the module instantiation.

```
interface simple_bus (input bit clk); // Define the interface
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
    logic start, rdy;


    modport slave (input req, addr, mode, start, clk,
                   output gnt, rdy,
                   ref data);

    modport master(input gnt, rdy, clk,
                   output req, addr, mode, start,
                   ref data);

endinterface: simple_bus

module memMod(simple_bus a); // Uses just the interface name
```

```
    logic avail;

    always @(posedge a.clk) // the clk signal from the interface
        a.gnt <= a.req & avail; // the gnt and req signal in the interface
endmodule

module cpuMod(simple_bus b);
    ...
endmodule

module top;
    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate the interface

    initial repeat(10) #10 clk++;

    memMod mem(sb_intf.slave); // Connect the modport to the module instance
    cpuMod cpu(sb_intf.master);
endmodule
```

20.4.3 Example of connecting port bundle to generic interface

This interface example shows how to use modports to control signal directions. It shows the use of the
interface keyword in the module definition. The actual interface and modport are specified in the module
instantiation.

```
    interface simple_bus (input bit clk); // Define the interface
        logic req, gnt;
        logic [7:0] addr, data;
        logic [1:0] mode;
        logic start, rdy;

        modport slave (input req, addr, mode, start, clk,
                       output gnt, rdy,
                       ref data);

        modport master(input gnt, rdy, clk,
                       output req, addr, mode, start,
                       ref data);

    endinterface: simple_bus

    module memMod(interface a); // Uses just the interface
        logic avail;

        always @(posedge a.clk) // the clk signal from the interface
            a.gnt <= a.req & avail; // the gnt and req signal in the interface
    endmodule

    module cpuMod(interface b);
        ...
    endmodule

    module top;
        logic clk = 0;

        simple_bus sb_intf(clk); // Instantiate the interface
```

```
        memMod mem(sb_intf.slave); // Connect the modport to the module instance
        cpuMod cpu(sb_intf.master);
    endmodule
```

20.4.4 Modport expressions

A modport expression allows elements of arrays and structures, concatenations of elements, and assignment pattern expressions of elements declared in an interface to be included in a modport list. This modport expression is explicitly named with a port identifier, visible only through the modport connection.

Like explicitly named ports in a module port declaration, port identifiers exist in their own name space for each modport list. When a modport item is just a simple port identifier, that identifier is used as both a reference to an interface item and a port identifier. Once a port identifier has been defined, there shall not be another port definition with this same name.

For example:

```
    interface I;
        logic [7:0] r;
        const int x=1;
        bit R;
        modport A (output .P(r[3:0]), input .Q(x), R);
        modport B (output .P(r[7:4]), input .Q(2), R);
    endinterface

    module M ( interface i);,
        initial i.P = i.Q;
    endmodule

    module top;
        I i1;
        M u1 (i1.A);
        M u2 (i1.B);
        initial #1 $display("%b", i1.r);    // displays 00010010
    endmodule
```

The self-determined type of the port expression becomes the type for the port. The port expression shall not be considered an assignment-like context. The port expression must resolve to a legal expression for type of module port (see 19.12). In the example above, the Q port could not be an output or inout because the port expression is a constant. The port expression is optional because ports can be defined that do not connect to anything internal to the port.

The following example illustrates how a bus with a parameterizable number of clients can be described:

```
    // Bus interface with parameterized number of client modports
    interface intf_t #(num_clients = 0);
        bit [num_clients-1:0] req;

        for (genvar i=0; i< num_clients; i++) begin: mps
          modport client_mp (output .client_req( req[i] ));
        end
    endinterface

    // A generic client that attaches to the bus
    module client_m (interface client_ifc);
        // ... code will drive client_ifc.client_req
```

```
    endmodule

// The bus system with N clients
module bus #(N = 0);
    intf_t #(.num_clients(N)) intf();

    for (genvar j=0; j < N; j++) begin: clients
        client_m client (.client_ifc (intf.mps[j].client_mp));
    end
endmodule
```

20.4.5 Clocking blocks and modports

The **modport** construct can also be used to specify the direction of **clocking** blocks declared within an interface. As with other **modport** declarations, the directions of the **clocking** block are those seen from the module in which the interface becomes a port. The syntax for this is shown below.

---

modport_declaration ::= **modport** modport_item { **,** modport_item } **;**   *// from A.2.9*

modport_item ::= modport_identifier **(** modport_ports_declaration { **,** modport_ports_declaration } **)**

modport_ports_declaration ::=
      { attribute_instance } modport_simple_ports_declaration
    | { attribute_instance } modport_tf_ports_declaration
    | { attribute_instance } modport_clocking_declaration

modport_clocking_declaration ::= **clocking** clocking_identifier

---

Syntax 20-2—Modport clocking declaration syntax (excerpt from Annex A)

All of the **clocking** blocks used in a **modport** declaration shall be declared by the same interface as the modport itself. Like all **modport** declarations, the direction of the clocking signals are those seen from the module in which the interface becomes a port. The example below shows how modports can be used to create both synchronous as well as asynchronous ports. When used in conjunction with virtual interfaces (see 20.8.2), these constructs facilitate the creation of abstract synchronous models.

```
    interface A_Bus( input bit clk );
        wire req, gnt;
        wire [7:0] addr, data;

        clocking sb @(posedge clk);
            input gnt;
            output req, addr;
            inout data;

            property p1; req ##[1:3] gnt; endproperty
        endclocking

        modport DUT ( input clk, req, addr,   // Device under test modport
                      output gnt,
                      inout data );

        modport STB ( clocking sb );          // synchronous testbench modport

        modport TB ( input gnt,               // asynchronous testbench modport
                     output req, addr,
                     inout data );
    endinterface
```

The above interface `A_Bus` can then be instantiated as shown below:

```
module dev1(A_Bus.DUT b);      // Some device: Part of the design
   ...
endmodule

module dev2(A_Bus.DUT b);      // Some device: Part of the design
   ...
endmodule

module top;
   bit clk;

   A_Bus b1( clk );
   A_Bus b2( clk );

   dev1 d1( b1 );
   dev2 d2( b2 );

   T tb( b1, b2 );
endmodule

program T (A_Bus.STB b1, A_Bus.STB b2 ); // testbench: 2 synchronous ports

   assert property (b1.p1);      // assert property from within program

   initial begin
      b1.sb.req <= 1;
      wait( b1.sb.gnt == 1 );
      ...
      b1.sb.req <= 0;
      b2.sb.req <= 1;
      wait( b2.sb.gnt == 1 );
      ...
      b2.sb.req <= 0;
   end
endprogram
```

The example above shows the program block using the synchronous interface designated by the clocking modport of interface ports `b1` and `b2`. In addition to the procedural drives and samples of the **clocking** block signals, the program asserts the property `p1` of one of its interfaces `b1`.

20.5 Interfaces and specify blocks

The **specify** block is used to describe various paths across a module and perform timing checks to ensure that events occurring at the module inputs satisfy the timing constraints of the device described by the module. The module paths are from module input ports to output ports, and the timing checks are relative to the module inputs. The **specify** block refers to these ports as terminal descriptor. Module **inout** ports can function as either an input or output terminal. When one of the port instances is an interface, each signal in the interface becomes an available terminal, with the default direction as defined for an interface or as restricted by a modport. A **ref** port cannot be used as a terminal in a **specify** block.

The following shows an example of using interfaces together with a **specify** block:

```
interface itf;
   logic c,q,d;
```

```
    modport flop (input c,d, output q);
endinterface

module dtype (itf.flop ch);
    always_ff @(posedge ch.c) ch.q <= ch.d;

    specify
       ( posedge ch.c => (ch.q+:ch.d)) = (5,6);
       $setup( ch.d, posedge ch.c, 1 );
    endspecify
endmodule
```

20.6 Tasks and functions in interfaces

Tasks and functions can be defined within an interface, or they can be defined within one or more of the modules connected. This allows a more abstract level of modeling. For example, "read" and "write" can be defined as tasks, without reference to any wires, and the master module can merely call these tasks. In a **modport**, these tasks are declared as **import** tasks.

A function prototype specifies the types and directions of the arguments and the return value of a function that is defined elsewhere. Similarly, a task prototype specifies the types and directions of the arguments of a task that is defined elsewhere. In a modport, the **import** and **export** constructs can either use function or task prototypes or use just the identifiers. The only exceptions are when a modport is used to import a function or task from another module and when default argument values or argument binding by name is used, in which cases a full prototype shall be used.

The number and types of arguments in a prototype must match the argument types in the function or task declaration. The rules for type matching are described in 6.9.1. If a default argument value is needed in a subroutine call, it shall be specified in the prototype. If an argument has default values specified in both the prototype and the declaration, the specified values need not be the same, but the default value used shall be the one specified in the prototype. Formal argument names in a prototype shall be optional unless default argument values or argument binding by name is used or additional unpacked dimensions are declared. The formal argument names in the prototype shall be the same as the formal argument names in a declaration.

If a module is connected to a modport containing an exported task or function and the module does not define that task or function, then an elaboration error shall occur. Similarly, if the modport contains an exported task or function prototype and the task or function defined in the module does not exactly match that prototype, then an elaboration error shall occur.

If the tasks or functions are defined in a module, using a hierarchical name, they must also be declared as **extern** in the interface or as **export** in a **modport**.

Tasks (not functions) can be defined in a module that is instantiated twice, e.g., two memories driven from the same central processing unit (CPU). Such multiple task definitions are allowed by an **extern fork-join** declaration in the interface.

20.6.1 Example of using tasks in interface

```
interface simple_bus (input bit clk); // Define the interface
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
    logic start, rdy;

    task masterRead(input logic [7:0] raddr); // masterRead method
```

```
        // ...
    endtask: masterRead

    task slaveRead; // slaveRead method
        // ...
    endtask: slaveRead

endinterface: simple_bus

module memMod(interface a); // Uses any interface
    logic avail;

    always @(posedge a.clk) // the clk signal from the interface
        a.gnt <= a.req & avail // the gnt and req signals in the interface

    always @(a.start)
        a.slaveRead;
endmodule

module cpuMod(interface b);
    enum {read, write} instr;
    logic [7:0] raddr;

    always @(posedge b.clk)
        if (instr == read)
            b.masterRead(raddr); // call the Interface method
        ...
endmodule

module top;
    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate the interface

    memMod mem(sb_intf);
    cpuMod cpu(sb_intf);
endmodule
```

20.6.2 Example of using tasks in modports

This interface example shows how to use modports to control signal directions and task access in a full read/
write interface.

```
interface simple_bus (input bit clk); // Define the interface
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
    logic start, rdy;

    modport slave (input req, addr, mode, start, clk,
                   output gnt, rdy,
                   ref data,
                   import slaveRead,
                           slaveWrite);
            // import into module that uses the modport

    modport master(input gnt, rdy, clk,
                   output req, addr, mode, start,
```

```
                            ref data,
                            import masterRead,
                                   masterWrite);
                    // import into module that uses the modport

            task masterRead(input logic [7:0] raddr); // masterRead method
               // ...
            endtask

            task slaveRead; // slaveRead method
               // ...
            endtask

            task masterWrite(input logic [7:0] waddr);
               //...
            endtask

            task slaveWrite;
               //...
            endtask

        endinterface: simple_bus

    module memMod(interface a); // Uses just the interface
        logic avail;

        always @(posedge a.clk) // the clk signal from the interface
           a.gnt <= a.req & avail; // the gnt and req signals in the interface

        always @(a.start)
           if (a.mode[0] == 1'b0)
              a.slaveRead;
           else
              a.slaveWrite;
    endmodule

    module cpuMod(interface b);
        enum {read, write} instr = $rand();
        logic [7:0] raddr = $rand();

        always @(posedge b.clk)
           if (instr == read)
              b.masterRead(raddr); // call the Interface method
           // ...
           else
              b.masterWrite(raddr);
    endmodule

    module omniMod( interface b);
        //...
    endmodule: omniMod

    module top;
        logic clk = 0;

        simple_bus sb_intf(clk); // Instantiate the interface

        memMod mem(sb_intf.slave); // only has access to the slave tasks
        cpuMod cpu(sb_intf.master); // only has access to the master tasks
```

```
      omniMod omni(sb_intf); // has access to all master and slave tasks
   endmodule
```

20.6.3 Example of exporting tasks and functions

This interface example shows how to define tasks in one module and call them in another, using modports to control task access.

```
   interface simple_bus (input bit clk); // Define the interface
      logic req, gnt;
      logic [7:0] addr, data;
      logic [1:0] mode;
      logic start, rdy;

      modport slave( input req, addr, mode, start, clk,
                     output gnt, rdy,
                     ref data,
                     export Read,
                            Write);
             // export from module that uses the modport

      modport master(input gnt, rdy, clk,
                     output req, addr, mode, start,
                     ref data,
                     import task Read(input logic [7:0] raddr),
                          task Write(input logic [7:0] waddr));
             // import requires the full task prototype

   endinterface: simple_bus

   module memMod(interface a); // Uses just the interface keyword
      logic avail;

      task a.Read; // Read method
         avail = 0;
         ...
         avail = 1;
      endtask

      task a.Write;
         avail = 0;
         ...
         avail = 1;
      endtask
   endmodule

   module cpuMod(interface b);
      enum {read, write} instr;
      logic [7:0] raddr;

      always @(posedge b.clk)
         if (instr == read)
            b.Read(raddr); // call the slave method via the interface
            ...
         else
            b.Write(raddr);
   endmodule
```

```
module top;
    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate the interface

    memMod mem(sb_intf.slave); // exports the Read and Write tasks
    cpuMod cpu(sb_intf.master); // imports the Read and Write tasks
endmodule
```

20.6.4 Example of multiple task exports

It is normally an error for more than one module to export the same task name. However, several instances of the same modport type can be connected to an interface, such as memory modules in the previous example. So that these can still export their read and write tasks, the tasks must be declared in the interface using the **extern forkjoin** keywords.

The call to extern forkjoin task countslaves( ); in the example below behaves as follows:

```
fork
    top.mem1.a.countslaves;
    top.mem2.a.countslaves;
join
```

For a read task, only one module should actively respond to the task call, e.g., the one containing the appropriate address. The tasks in the other modules should return with no effect. Only then should the active task write to the result variables.

Unlike tasks, multiple export of functions is not allowed because they must always write to the result.

The effect of a **disable** on an extern forkjoin task is as follows:
— If the task is referenced via the interface instance, all task calls shall be disabled.
— If the task is referenced via the module instance, only the task call to that module instance shall be disabled.
— If an interface contains an extern forkjoin task and no module connected to that interface defines the task, then any call to that task shall report a run-time error and return immediately with no effect.

This interface example shows how to define tasks in more than one module and call them in another using **extern forkjoin**. The multiple task export mechanism can also be used to count the instances of a particular modport that are connected to each interface instance.

```
interface simple_bus (input bit clk); // Define the interface
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
    logic start, rdy;
    int slaves = 0;

    // tasks executed concurrently as a fork/join block
    extern forkjoin task countSlaves();
    extern forkjoin task Read (input logic [7:0] raddr);
    extern forkjoin task Write (input logic [7:0] waddr);

    modport slave (input req,addr, mode, start, clk,
                   output gnt, rdy,
                   ref data, slaves,
```

```
                    export Read, Write, countSlaves);
               // export from module that uses the modport


      modport master (  input gnt, rdy, clk,
                        output req, addr, mode, start,
                        ref data,
                        import task Read(input logic [7:0] raddr),
                        task Write(input logic [7:0] waddr));
               // import requires the full task prototype


      initial begin
         slaves = 0;
         countSlaves;
         $display ("number of slaves = %d", slaves);
      end

   endinterface: simple_bus

   module memMod #(parameter int minaddr=0, maxaddr=0;) (interface a);
      logic avail = 1;
      logic [7:0] mem[255:0];

      task a.countSlaves();
         a.slaves++;
      endtask

      task a.Read(input logic [7:0] raddr); // Read method
         if (raddr >= minaddr && raddr <= maxaddr) begin
            avail = 0;
            #10 a.data = mem[raddr];
            avail = 1;
         end
      endtask

      task a.Write(input logic [7:0] waddr); // Write method
         if (waddr >= minaddr && waddr <= maxaddr) begin
            avail = 0;
            #10 mem[waddr] = a.data;
            avail = 1;
         end
      endtask
   endmodule

   module cpuMod(interface b);
      typedef enum {read, write} instr;
      instr inst;
      logic [7:0] raddr;
      integer seed;

      always @(posedge b.clk) begin
         inst = instr'($dist_uniform(seed, 0, 1));
         raddr = $dist_uniform(seed, 0, 3);
         if (inst == read) begin
            $display("%t begin read %h @ %h", $time, b.data, raddr);
            callr:b.Read(raddr);
            $display("%t end read %h @ %h", $time, b.data, raddr);
         end
         else begin
            $display("%t begin write %h @ %h", $time, b.data, raddr);
```

```
            b.data = raddr;
            callw:b.Write(raddr);
            $display("%t end write %h @ %h", $time, b.data, raddr);
        end
    end
endmodule

module top;
    logic clk = 0;

    function void interrupt();
        disable mem1.a.Read; // task via module instance
        disable sb_intf.Write; // task via interface instance
        if (mem1.avail == 0) $display ("mem1 was interrupted");
        if (mem2.avail == 0) $display ("mem2 was interrupted");
    endfunction

    always #5 clk++;

    initial begin
        #28 interrupt();
        #10 interrupt();
        #100 $finish;
    end

    simple_bus sb_intf(clk);

    memMod #(0, 127) mem1(sb_intf.slave);
    memMod #(128, 255) mem2(sb_intf.slave);
    cpuMod cpu(sb_intf.master);
endmodule
```

20.7 Parameterized interfaces

Interface definitions can take advantage of parameters and parameter redefinition, in the same manner as module definitions. This example shows how to use parameters in interface definitions.

```
    interface simple_bus #(AWIDTH = 8, DWIDTH = 8)
                        (input bit clk); // Define the interface
    logic req, gnt;
    logic [AWIDTH-1:0] addr;
    logic [DWIDTH-1:0] data;
    logic [1:0] mode;
    logic start, rdy;

    modport slave( input req, addr, mode, start, clk,
                    output gnt, rdy,
                    ref data,
                    import task slaveRead,
                            task slaveWrite);
            // import into module that uses the modport

    modport master(input gnt, rdy, clk,
                    output req, addr, mode, start,
                    ref data,
                    import task masterRead(input logic [AWIDTH-1:0] raddr),
                            task masterWrite(input logic [AWIDTH-1:0] waddr));
            // import requires the full task prototype
```

```
    task masterRead(input logic [AWIDTH-1:0] raddr); // masterRead method
        ...
    endtask

    task slaveRead; // slaveRead method
        ...
    endtask

    task masterWrite(input logic [AWIDTH-1:0] waddr);
        ...
    endtask

    task slaveWrite;
        ...
    endtask

endinterface: simple_bus

module memMod(interface a); // Uses just the interface keyword
    logic avail;

    always @(posedge a.clk) // the clk signal from the interface
        a.gnt <= a.req & avail; //the gnt and req signals in the interface

    always @(a.start)
        if (a.mode[0] == 1'b0)
            a.slaveRead;
        else
            a.slaveWrite;
endmodule

module cpuMod(interface b);
    enum {read, write} instr;
    logic [7:0] raddr;

    always @(posedge b.clk)
        if (instr == read)
            b.masterRead(raddr); // call the Interface method
            // ...
        else
            b.masterWrite(raddr);
endmodule

module top;

    logic clk = 0;

    simple_bus sb_intf(clk); // Instantiate default interface
    simple_bus #(.DWIDTH(16)) wide_intf(clk); // Interface with 16-bit data

    initial repeat(10) #10 clk++;

    memMod mem(sb_intf.slave); // only has access to the slaveRead task
    cpuMod cpu(sb_intf.master); // only has access to the masterRead task

    memMod memW(wide_intf.slave); // 16-bit wide memory
    cpuMod cpuW(wide_intf.master); // 16-bit wide cpu
endmodule
```

20.8 Virtual interfaces

Virtual interfaces provide a mechanism for separating abstract models and test programs from the actual signals that make up the design. A virtual interface allows the same subprogram to operate on different portions of a design and to dynamically control the set of signals associated with the subprogram. Instead of referring to the actual set of signals directly, users are able to manipulate a set of virtual signals. Changes to the underlying design do not require the code using virtual interfaces to be rewritten. By abstracting the connectivity and functionality of a set of blocks, virtual interfaces promote code reuse.

A virtual interface is a variable that represents an interface instance. The syntax to declare a virtual interface variable is given below.

virtual_interface_declaration ::=	*// from A.2.9*
virtual [**interface**] interface_identifier list_of_virtual_interface_decl **;**	
list_of_virtual_interface_decl ::=	*// from A.2.3*
variable_identifier [**=** interface_instance_identifier]	
{ **,** variable_identifier [**=** interface_instance_identifier] }	
data_declaration[14] ::=	*// from A.2.1.3*
...	
| virtual_interface_declaration	
data_type ::=	*// from A.2.2.1*
...	
| **virtual** [**interface**] interface_identifier	

Syntax 20-3—Virtual interface declaration syntax (excerpt from Annex A)

Virtual interface variables can be passed as arguments to tasks, functions, or methods. A single virtual interface variable can thus represent different interface instances at different times throughout the simulation. A virtual interface must be initialized before it can be used; it has the value **null** before it is initialized. Attempting to use an uninitialized virtual interface shall result in a fatal run-time error.

Only the following operations are directly allowed on virtual interface variables:
— Assignment (=) to the following:
  — Another virtual interface of the same type
  — An interface instance of the same type
  — The special constant **null**
— Equality (==) and inequality (!=) with the following:
  — Another virtual interface of the same type
  — An interface instance of the same type
  — The special constant **null**

Virtual interfaces shall not be used as ports, interface items, or as members of unions.

Once a virtual interface has been initialized, all the components of the underlying interface instance are directly available to the virtual interface via the dot notation. These components can only be used in procedural statements; they cannot be used in continuous assignments or sensitivity lists. In order for a net to be driven via a virtual interface, the interface itself must provide a procedural means to do so. This can be accomplished either via a **clocking** block or by including a driver that is updated by a continuous assignment from a variable within the interface.

Virtual interfaces can be declared as class properties, which can be initialized procedurally or by an argument to **new**(). This allows the same virtual interface to be used in different classes. The following example shows how the same transactor class can be used to interact with various different devices:

```
interface SBus;                      // A Simple bus interface
    logic req, grant;
    logic [7:0] addr, data;
endinterface


class SBusTransctor;                 // SBus transactor class
    virtual SBus bus;                // virtual interface of type Sbus

    function new( virtual SBus s );
        bus = s;                     // initialize the virtual interface
    endfunction

    task request();                  // request the bus
        bus.req <= 1'b1;
    endtask

    task wait_for_bus();             // wait for the bus to be granted
        @(posedge bus.grant);
    endtask
endclass

module devA( Sbus s ) ... endmodule  // devices that use SBus
module devB( Sbus s ) ... endmodule

module top;

    SBus s[1:4] ();                  // instantiate 4 interfaces

    devA a1( s[1] );                 // instantiate 4 devices
    devB b1( s[2] );
    devA a2( s[3] );
    devB b2( s[4] );

    initial begin
        SbusTransactor t[1:4];       // create 4 bus-transactors and bind

        t[1] = new( s[1] );
        t[2] = new( s[2] );
        t[3] = new( s[3] );
        t[4] = new( s[4] );
        // test t[1:4]
    end
endmodule
```

In the preceding example, the transaction class SbusTransctor is a simple reusable component. It is written without any global or hierarchical references and is unaware of the particular device with which it will interact. Nevertheless, the class can interact with any number of devices (four in the example) that adhere to the interface's protocol.

20.8.1 Virtual interfaces and clocking blocks

Interfaces and **clocking** blocks can be combined to represent the interconnect between synchronous blocks. Moreover, because **clocking** blocks provide a procedural mechanism to assign values to both nets and variables, they are ideally suited to be used by virtual interfaces. For example:

```
interface SyncBus( input bit clk );
    wire a, b, c;

    clocking sb @(posedge clk);
        input a;
        output b;
        inout c;
    endclocking

endinterface

typedef virtual SyncBus VI;      // A virtual interface type

task do_it( VI v );              // handles any SyncBus via clocking sb
    if( v.sb.a == 1 )
        v.sb.b <= 0;
    else
        v.sb.c <= ##1 1;
endtask
```

In the preceding example, interface `SyncBus` includes a **clocking** block, which is used by task `do_it` to ensure synchronous access to the interface's signals: `a`, `b`, and `c`. A change to the storage type of the interface signals (from net to variable and vice versa) requires no changes to the task. The interfaces can be instantiated as shown below.

```
module top;
    bit clk;

    SyncBus b1( clk );
    SyncBus b2( clk );

    initial begin
        VI v[2] = '{ b1, b2 };

        repeat( 20 )
            do_it( v[ $urandom_range( 0, 1 ) ] );
    end
endmodule
```

The top module above shows how a virtual interface can be used to randomly select among a set of interfaces to be manipulated, in this case by the `do_it` task.

20.8.2 Virtual interfaces modports and clocking blocks

As shown in the example above, once a virtual interface is declared, its **clocking** block can be referenced using dot notation. However, this only works for interfaces with no modports. Typically, a DUT and its testbench exhibit modport direction. This common case can be handled by including the clocking in the corresponding modport as described in 20.4.5.

The example below shows how modports used in conjunction with virtual interfaces facilitate the creation of abstract synchronous models.

```
interface A_Bus( input bit clk );
   wire req, gnt;
   wire [7:0] addr, data;

   clocking sb @(posedge clk);
      input gnt;
      output req, addr;
      inout data;

      property p1; req ##[1:3] gnt; endproperty
   endclocking

   modport DUT ( input clk, req, addr,    // Device under test modport
                 output gnt,
                 inout data );

   modport STB ( clocking sb );           // synchronous testbench modport

   modport TB ( input gnt,                // asynchronous testbench modport
                output req, addr,
                inout data );
endinterface
```

The above interface A_Bus can then be instantiated as shown below:

```
module dev1(A_Bus.DUT b);              // Some device: Part of the design
   ...
endmodule

module dev2(A_Bus.DUT b);              // Some device: Part of the design
   ...
endmodule

program T (A_Bus.STB b1, A_Bus.STB b2 ); // Testbench: 2 synchronous ports
      ...
endprogram

module top;
   bit clk;

   A_Bus b1( clk );
   A_Bus b2( clk );

   dev1 d1( b1 );
   dev2 d2( b2 );

   T tb( b1, b2 );
endmodule
```

And, within the testbench program, the virtual interface can refer directly to the **clocking** block.

```
program T (A_Bus.STB b1, A_Bus.STB b2 ); // Testbench: 2 synchronous ports

   typedef virtual A_Bus.STB SYNCTB;

   task request( SYNCTB s );
      s.sb.req <= 1;
```

```
            endtask

            task wait_grant( SYNCTB s );
               wait( s.sb.gnt == 1 );
            endtask

            task drive(SYNCTB s, logic [7:0] adr, data );
               if( s.sb.gnt == 0 ) begin
                  request(s);                    // acquire bus if needed
                  wait_grant(s);
               end
               s.sb.addr = adr;
               s.sb.data = data;
               repeat(2) @s.sb;
               s.sb.req = 0;                     //release bus
            endtask

            assert property (b1.p1);             // assert property from within program

            initial begin
               drive( b1, $random, $random );
               drive( b2, $random, $random );
            end
         endprogram
```

The example above shows how the `clocking` block is referenced via the virtual interface by the tasks within the program block.

20.9 Access to interface objects

Access to all objects declared in an interface is always available by hierarchical reference, regardless of whether the interface is connected through a port. When an interface is connected with a modport in either the module header or port connection, access by port reference is limited to only objects listed in the modport, for only types of objects legal to be listed in modports (nets, variables, tasks, and functions). All objects are still visible by hierarchical reference. For example:

```
   interface ebus_i;
      integer I;                  // reference to I not allowed through modport mp
      typedef enum {Y,N} choice;
      choice Q;
      localparam True = 1;
      modport mp(input Q);
   endinterface

   module Top;
      ebus_i ebus;
      sub s1(ebus.mp);
   endmodule

   module sub(interface.mp i);
      typedef i.choice yes_no;   // import type from interface
      yes_no P;
      assign P = i.Q;            // refer to Q with a port reference
      initial
         Top.ebus.Q = i.True;    // refer to Q with a hierarchical reference
      initial
         Top.ebus.I = 0;         // referring to i.I would not be legal because
```

```
                                          // is not in modport mp
endmodule
```

21. Configuration libraries

21.1 Introduction

NOTE—Verilog provides the ability to specify design configurations, which specify the binding information of module instances to specific Verilog HDL source code. Configurations utilize libraries. A library is a collection of modules, primitives, and other configurations. Separate library map files specify the source code location for the cells contained within the libraries. The names of the library map files are typically specified as invocation options to simulators or other software tools reading in Verilog source code.

SystemVerilog adds support for several new constructs to Verilog configurations.

21.2 Libraries

A library is a named collection of cells. A cell is a module, macromodule, primitive, interface, program, package, or configuration. A configuration is a specification of which source files bind to each instance in the design. A configuration may change the binding of a module, macromodule, primitive, interface, or program instance, but shall not change the binding of a package.

22. System tasks and system functions

22.1 Introduction

NOTE—SystemVerilog adds several system tasks and system functions, as described in this clause.

In addition, SystemVerilog extends the behavior of several Verilog system tasks, as described in 22.13.

22.2 Type name function

typename_function ::=	// *not in* Annex A
$typename (expression **)**	
| **$typename (** data_type **)**	

Syntax 22-1—Type name function syntax (not in Annex A)

The `$typename` system function returns a string that represents the resolved type of its argument.

The return string is constructed in the following steps:

  a)   A **typedef** that creates an equivalent type is resolved back to built-in or user-defined types.

  b)   The default signing is removed, even if present explicitly in the source.

  c)   System-generated names are created for anonymous structs, unions, and enums.

  d)   A '$' is used as the placeholder for the name of an anonymous unpacked array.

  e)   Actual encoded values are appended with enumeration named constants.

  f)   User-defined type names are prefixed with their defining package or scope name space.

  g)   Array ranges are represented as unsized decimal numbers.

  h)   Whitespace in the source is removed and a single space is added to separate identifiers and keywords from each other.

This process is similar to the way that type matching (see 6.9.1) is computed, except that simple bit vectors types with predefined widths are distinguished from those with user-defined widths. Thus `$typename` can be used in string comparisons for stricter type comparison of arrays than with type references.

When called with an expression as its argument, `$typename` returns a string that represents the self-determined type result of the expression. The expression's return type is determined during elaboration, but never evaluated. When used as an elaboration time constant, the expression shall not contain any hierarchical references or references to elements of dynamic objects.

```
// source code                        // $typename would return
typedef bit node;                     // "bit"
node signed [2:0] X;                  // "bit signed[2:0]"
int signed Y;                         // "int"
package A;
    enum {A,B,C=99} X;                // "enum{A=32'd0,B=32'd1,C='32d99}A::e$1"
    typedef bit [9:1'b1] word;        // "A::bit[9:1]"
endpackage : A
import A::*;
module top;
    typedef struct {node A,B;} AB_t;
    AB_t AB[10];                      // "struct{bit A;bit B;}top.AB_t$[0:9]"
    ...
endmodule
```

22.3 Expression size system function

```
size_function ::=                                                    // not in Annex A
        $bits ( expression )
      | $bits ( data_type )
```

Syntax 22-2—Size function syntax (not in Annex A)

The `$bits` system function returns the number of bits required to hold an expression as a bit stream. See 4.16 for a definition of legal types. A 4-state value counts as 1 bit. Given the declaration:

```
logic [31:0] foo;
```

then `$bits(foo)` shall return 32, even if the implementation uses more than 32 bits of storage to represent the 4-state values. Given the declaration:

```
typedef struct {
   logic valid;
   bit [8:1] data;
} MyType;
```

the expression `$bits(MyType)` shall return 9, the number of data bits needed by a variable of type `MyType`.

The `$bits` function can be used as an elaboration time constant when used on fixed-size data types; hence, it can be used in the declaration of other data types, variables, or nets.

```
typedef bit[$bits(MyType):1] MyBits;   //same as typedef bit [9:1] MyBits;
MyBits b;
```

Variable `b` can be used to hold the bit pattern of a variable of type `MyType` without loss of information.

The value returned by `$bits` shall be determined without actual evaluation of the expression it encloses. It shall be an error to enclose a function that returns a dynamically sized data type. The `$bits` return value shall be valid at elaboration only if the expression contains fixed-size data types.

The `$bits` system function returns 0 when called with a dynamically sized expression that is currently empty. It shall be an error to use the `$bits` system function directly with a dynamically sized data type identifier.

22.4 Range system function

```
range_function ::=                                                   // not in Annex A
        $isunbounded ( constant_expression )
```

Syntax 22-3—Range function syntax (not in Annex A)

The `$isunbounded` system function returns true if the argument is `$`. Given the declaration:

```
parameter int foo = $;
```

then `$isunbounded(foo)` shall return true. Otherwise, it shall return false. True and false are defined in 22.9.

22.5 Shortreal conversions

Verilog defines a **real** data type and the system functions $realtobits and $bitstoreal to permit exact bit pattern transfers between a **real** and a 64-bit vector. SystemVerilog adds the **shortreal** type, and in a parallel manner, $shortrealtobits and $bitstoshortreal are defined to permit exact bit transfers between a **shortreal** and a 32-bit vector.

```
[31:0] $shortrealtobits(shortreal_val) ;
shortreal $bitstoshortreal(bit_val) ;
```

$shortrealtobits converts from a **shortreal** number to the 32-bit representation (vector) of that **shortreal** number. $bitstoshortreal is the reverse of $shortrealtobits; it converts from the bit pattern to a **shortreal** number.

22.6 Array querying system functions

```
array_query_function ::=                                                // not in Annex A
        array_dimension_function ( array_identifier [ , dimension_expression ] )
      | array_dimension_function ( data_type [ , dimension_expression ] )
      | array_dimensions_function ( array_identifier )
      | array_dimensions_function ( data_type )
array_dimensions_function ::=
        $dimensions
      | $unpacked_dimensions
array_dimension_function ::=
        $left
      | $right
      | $low
      | $high
      | $increment
      | $size
dimension_expression ::= expression
```

Syntax 22-4—Array querying function syntax (not in Annex A)

SystemVerilog provides system functions to return information about a particular dimension of an array (see Clause 5) or integral (see 4.3.1) data type or of data objects of such a data type.

SystemVerilog provides system functions to return information about a particular dimension of an array data type or object (see Clause 4) or integral data type or object (see 4.3.1).

The return type is **integer**, and the default for the optional dimension expression is 1. The array dimension can specify any fixed-size index (packed or unpacked) or any dynamically sized index (dynamic, associative, or queue).

— $left shall return the left bound (MSB) of the dimension.
— $right shall return the right bound (LSB) of the dimension.
— $low shall return the minimum of $left and $right of the dimension.
— $high shall return the maximum of $left and $right of the dimension.
— $increment shall return 1 if $left is greater than or equal to $right and −1 if $left is less than $right.

— `$size` shall return the number of elements in the dimension, which is equivalent to `$high − $low +` 1.

— `$dimensions` shall return the following:

— The total number of dimensions in the array (packed and unpacked, static or dynamic)

— 1 for the **string** data type or any other nonarray type that is equivalent to a simple bit vector type (see 4.3.1)

— 0 for any other type

— `$unpacked_dimensions` shall return the following:

— The total number of unpacked dimensions for an array (static or dynamic)

— 0 for any other type

The dimensions of an array shall be numbered as follows: The slowest varying dimension (packed or unpacked) is dimension 1. Successively faster varying dimensions have sequentially higher dimension numbers. Intermediate type definitions are expanded first before numbering the dimensions.

For example:

```
//      Dimension numbers
//    3    4      1    2
reg [3:0][2:1] n [1:5][2:8];
typedef reg [3:0][2:1] packed_reg;
packed_reg n[1:5][2:8]; // same dimensions as in the lines above
```

For a fixed-size integer type (**integer**, **shortint**, **longint**, and **byte**), dimension 1 is predefined. For an integer *N* declared without a range specifier, its bounds are assumed to be `[$bits(N)-1:0]`.

If the first argument to an array query function would cause `$dimensions` to return 0 or if the second argument is out of range, then an `'x` shall be returned.

When used on a dynamic array or queue dimension, these functions return information about the current state of the array. If the dimension is currently empty, these functions shall return a `'x`. It is an error to use these functions directly on a dynamically sized type identifier.

Use on associative array dimensions is restricted to index types with integral values. With integral indexes, these functions shall return the following:

— `$left` shall return 0.

— `$right` shall return the highest possible index value.

— `$low` shall return the lowest currently allocated index value.

— `$high` shall return the largest currently allocated index value.

— `$increment` shall return −1.

— `$size` shall return the number of elements currently allocated.

If the array identifier is a fixed-size array, these query functions can be used as a constant function and passed as a parameter before elaboration. These query functions can also be used on fixed-size type identifiers, in which case it is always treated as a constant function.

Given the declaration:

```
typedef logic [16:1] Word;
Word Ram[0:9];
```

the following system functions return 16:

```
$size(Word)
$size(Ram,2)
```

22.7 Assertion severity system tasks

assert_severity_task ::=   *// not in* Annex A
fatal_message_task
| nonfatal_message_task
fatal_message_task ::= **$fatal** [**(** finish_number [**,** message_argument { **,** message_argument }] **)**] **;**
nonfatal_message_task ::= severity_task [**(** [message_argument { **,** message_argument] }] **)**] **;**
severity_task ::= **$error** \| **$warning** | **$info**
finish_number ::= **0** | **1** | **2**
message_argument ::= string | expression

Syntax 22-5—Assertion severity system task syntax (not in Annex A)

SystemVerilog assertions have a severity level associated with any assertion failures detected. By default, the severity of an assertion failure is "error". The severity levels can be specified by including one of the following severity system tasks in the assertion fail statement:

— $fatal shall generate a run-time fatal assertion error, which terminates the simulation with an error code. The first argument passed to $fatal shall be consistent with the corresponding argument to the Verilog $finish system task, which sets the level of diagnostic information reported by the tool. Calling $fatal results in an implicit call to $finish.

— $error shall be a run-time error.

— $warning shall be a run-time warning, which can be suppressed in a tool-specific manner.

— $info shall indicate that the assertion failure carries no specific severity.

All of these severity system tasks shall print a tool-specific message, indicating the severity of the failure, and specific information about the failure, which shall include the following information:

— The file name and line number of the assertion statement

— The hierarchical name of the assertion if it is labeled or the scope of the assertion if it is not labeled.

For simulation tools, these tasks shall also report the simulation run time at which the severity system task is called.

Each of the severity tasks can include optional user-defined information to be reported. The user-defined message shall use the same syntax as the Verilog **$display** system task and can include any number of arguments.

22.8 Assertion control system tasks

```
assert_control_task ::=                                                    // not in Annex A
        assert_task [ ( levels [ , list_of_modules_or_assertions ] ) ] ;
assert_task ::=
        $asserton
      | $assertoff
      | $assertkill
list_of_modules_or_assertions ::=
        module_or_assertion { , module_or_assertion }
module_or_assertion ::=
        module_identifier
      | assertion_identifier
      | hierarchical_identifier
```

Syntax 22-6—Assertion control syntax (not in Annex A)

SystemVerilog provides three system tasks to control assertions.

— `$assertoff` shall stop the checking of all specified assertions until a subsequent `$asserton`. An assertion that is already executing, including execution of the pass or fail statement, is not affected.

— `$assertkill` shall abort execution of any currently executing specified assertions and then stop the checking of all specified assertions until a subsequent `$asserton`.

— `$asserton` shall reenable the execution of all specified assertions.

When invoked with no arguments, the system task shall apply to all assertions. When the task is specified with arguments, the first argument indicates levels of the hierarchy, consistent with the corresponding argument to the Verilog `$dumpvars` system task. Subsequent arguments specify which scopes of the model to control. These arguments can specify entire modules or individual assertions.

22.9 Assertion system functions

```
assert_boolean_functions ::=                                               // not in Annex A
        assert_function ( expression ) ;
assert_function ::=
        $onehot
      | $onehot0
      | $isunknown
```

Syntax 22-7—Assertion system function syntax (not in Annex A)

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is "one-hot". The following system functions are included to facilitate such common assertion functionality:

— `$onehot` returns true if 1 and only 1 bit of expression is high.

— `$onehot0` returns true if at most 1 bit of expression is high.

— `$isunknown` returns true if any bit of the expression is X or Z. This is equivalent to `^expression === 'bx`.

All of the above system functions shall have a return type of **bit**. A return value of 1'b1 shall indicate true, and a return value of 1'b0 shall indicate false.

A function is provided to return sampled value of an expression.

```
$sampled ( expression [, clocking_event] )
```

Three functions are provided for assertions to detect changes in values between two adjacent clock ticks.

```
$rose ( expression [, clocking_event] )

$fell ( expression [, clocking_event] )

$stable ( expression [, clocking_event] )
```

The past values can be accessed with the $past function.

```
$past ( expression [, number_of_ticks] [, expression2] [, clocking_event] )
```

Functions $sampled, $rose, $fell, $stable, and $past are discussed in 17.7.3.

The number of ones in a bit vector expression can be determined with the $countones function.

```
$countones ( expression )
```

$countones is discussed in 17.10.

22.10 Random number system functions

To supplement the Verilog $random system function, SystemVerilog provides two special system functions for generating pseudo-random numbers, $urandom and $urandom_range. These system functions are presented in 13.12.

22.11 Program control

In addition to the normal simulation control tasks ($stop and $finish), a program can use the $exit control task. When all programs exit, the simulation finishes and an implicit call to $finish is made. The usage of $exit is presented in 16.6 on program blocks.

22.12 Coverage system functions

SystemVerilog has several built-in system functions for obtaining test coverage information: $coverage_control, $coverage_get_max, $coverage_get, $coverage_merge, and $coverage_save. The coverage system functions are described in 29.2.

22.13 Enhancements to Verilog system tasks

SystemVerilog adds system tasks and system functions as described in 22.14 and 22.15. In addition, System-Verilog extends the behavior of the following:

— $display, $write, $fdisplay, $fwrite, $swrite, and their variants

  — The format arguments to these tasks must be string literals, i.e., they cannot be expressions of **string** data type. The only exception is $sformat, whose second argument can be an expression of **string** data type.

  — The first argument of $swrite can be a **string** variable.

  — The integer % format specifiers (h, d, o, b, c, u, and z) may be used with any of the SystemVerilog integral data types, including enumerated types and packed aggregate data types. They shall not be used with any unpacked aggregate type.

  — The argument corresponding to a string % format specifier (s) may have the **string** data type.

  — The real number % format specifiers (e, f, and g) may be used with the **shortreal** data type.

  — The above format specifiers can also be used with user-defined types that have been defined (using **typedef**) to be represented using one of these basic types.

— $fscanf and $sscanf

  — The format arguments to these tasks may be expressions of **string** data type.

  — The first argument of $sscanf can be a **string** variable.

  — The integer % format specifiers (b, o, d, and h) may be used to read into any of the SystemVerilog integral data types, including enumerated types and packed aggregate data types. They shall not be used with any unpacked aggregate type.

  — The string % format specifier (s) may be used to read into variables of the **string** data type.

  — The real number % format specifiers (e, f, and g) may be used with the **shortreal** data type.

  — The above format specifiers can also be used with user-defined types that have been defined (using **typedef**) to be represented using one of these basic types.

— %u and %z format specifiers

  — For packed data, %u and %z are defined to operate as though the operation were applied to the equivalent vector.

  — For unpacked **struct** data, %u and %z are defined to apply as though the operation were performed on each member in declaration order.

  — For unpacked **union** data, %u and %z are defined to apply as though the operation were performed on the first member in declaration order.

  — %u and %z are not defined on unpacked arrays.

  — The count of data items read by a %u or %z for an aggregate type is always either 1 or 0; the individual members are not counted separately.

— $fread, which has two variants: a register variant and a set of three memory variants

  — The register variant,

    ```
    $fread(myreg, fd);
    ```

  is defined to be the one applied for all packed data.

  — For unpacked **struct** data, $fread is defined to apply as though the operation were performed on each member in declaration order.

  — For unpacked **union** data, $fread is defined to apply as though the operation were performed on the first member in declaration order.

— For unpacked arrays, the original definition applies except that unpacked **struct** or **union** elements are read as described above.

22.14 $readmemb and $readmemh

22.14.1 Reading packed data

$readmemb and $readmemh are extended to unpacked arrays of packed data, associative arrays of packed data, and dynamic arrays of packed data. In such cases, the system tasks treat each packed element as the vector equivalent and perform the normal operation.

When working with associative arrays, indexes must be of integral types. When an associative array's index is of an enumerated type, address entries in the pattern file are in numeric format and correspond to the numeric values associated with the elements of the enumerated type.

22.14.2 Reading 2-state types

$readmemb and $readmemh are extended to packed data of 2-state types, such as **int** or enumerated types. For 2-state integer types, reading proceeds the same as for conventional Verilog variable types (e.g., integer), with the exception that X or Z data are converted to 0. For enumerated types, the file data represents the numeric values associated with each element of the enumerated type (see 4.10). If a numeric value is out of range for a given type, then an error shall be issued and no further reading shall take place.

22.15 $writememb and $writememh

SystemVerilog introduces system tasks $writememb and $writememh:

---

writemem_tasks ::=   // *not in* Annex A
    $writememb (" file_name **" ,** memory_name [**,** start_addr [**,** finish_addr]] **) ;**
    | **$writememh ("** file_name **" ,** memory_name [**,** start_addr [**,** finish_addr]] **) ;**

Syntax 22-8—Writemem system task syntax (not in Annex A)

$writememb and $writememh are used to dump memory contents to files that are readable by $readmemb and $readmemh, respectively. If "file_name" exists at the time $writememb or $writememh is called, the file will be overwritten (.i.e., there is no append mode).

22.15.1 Writing packed data

$writememb and $writememh treat packed data identically to $readmemb and $readmemh. See 22.14.1

22.15.2 Writing 2-state types

$writememb and $writememh can write out data corresponding to unpacked arrays of 2-state types, such as **int** or enumerated types. For enumerated types, values in the file correspond to the ordinal values of the enumerated type (see 4.10).

22.15.3 Writing addresses to output file

When $writememb and $writememh write out data corresponding to unpacked or dynamic arrays, address specifiers (@-words) shall not be written to the output file.

When `$writememb` and `$writememh` write out data corresponding to associative arrays, address specifiers shall be written to the output file. As specified in 22.14.1, associative arrays shall have indexes of integral types in order to be legal arguments to the `$writememb` and `$writememh` calls.

22.16 File format considerations for multidimensional unpacked arrays

In SystemVerilog, `$readmemb`, `$readmemh`, `$writememb`, and `$writememh` can work with multidimensional unpacked arrays.

The file contents are organized in row-major order, with each dimension's entries ranging from low to high address. This is backward compatible with plain Verilog memories.

In this organization, the lowest dimension (i.e., the rightmost dimension in the array declaration) varies the most rapidly. There is a hierarchical sense to the file data. The higher dimensions contain words of lower dimension data, sorted in row-major order. Each successive lower dimension is entirely enclosed as part of higher dimension words.

As an example of file format organization, here is the layout of a file representing words for a memory declared:

```
reg [31:0] mem [0:2][0:4][5:8];
```

In the example word contents, `wzyx`,

— `z` corresponds to words of the `[0:2]` dimension.
— `y` corresponds to words of the `[0:4]` dimension.
— `x` corresponds to words of the `[5:8]` dimension.

```
w005 w006 w007 w008
w015 w016 w017 w018
w025 w026 w027 w028
w035 w036 w037 w038
w045 w046 w047 w048
w105 w106 w107 w108
w115 w116 w117 w118
w125 w126 w127 w128
w135 w136 w137 w138
w145 w146 w147 w148
w205 w206 w207 w208
w215 w216 w217 w218
w225 w226 w227 w228
w235 w236 w237 w238
w245 w246 w247 w248
```

The above diagram would be identical if one or more of the unpacked dimension declarations were reversed, as in:

```
reg [31:0] mem [2:0][0:4][8:5]
```

Address entries in the file exclusively address the highest dimension's words. In the above case, address entries in the file could look something as follows:

```
@0 w005 w006 w007 w008
   w015 w016 w017 w018
   w025 w026 w027 w028
```

```
    w035 w036 w037 w038
    w045 w046 w047 w048
@1  w105 w106 w107 w108
    w115 w116 w117 w118
    w125 w126 w127 w128
    w135 w136 w137 w138
    w145 w146 w147 w148
@2  w205 w206 w207 w208
    w215 w216 w217 w218
    w225 w226 w227 w228
    w235 w236 w237 w238
    w245 w246 w247 w248
```

When $readmemh or $readmemb is given a file without address entries, all data are read assuming that each dimension has complete data. i.e., each word in each dimension will be initialized with the appropriate value from the file. If the file contains incomplete data, the read operation will stop at the last initialized word, and any remaining array words or subwords will be left unchanged.

When $readmemh or $readmemb is given a file with address entries, initialization of the specified highest dimension words is done. If the file contains insufficient words to completely fill a highest dimension word, then the remaining subwords are left unchanged.

When a memory contains multiple packed dimensions, the memory words in the pattern file are composed of the sum total of all bits in the packed dimensions. The layout of packed bits in packed dimensions is defined in 5.3.

22.17 System task arguments for multidimensional unpacked arrays

The $readmemb, $readmemh, $writememb, and $writememh signatures are shown below:

```
$readmemb("file_name", memory_name[, start_addr[, finish_addr]]);
$readmemh("file_name", memory_name[, start_addr[, finish_addr]]);
$writememb("file_name", memory_name[, start_addr[, finish_addr]]);
$writememh("file_name", memory_name[, start_addr[, finish_addr]]);
```

The *memory_name* can be an unpacked array or a partially indexed multidimensional unpacked array that resolves to a lesser dimensioned unpacked array.

Higher order dimensions must be specified with an index, rather than a complete or partial dimension range. The lowest dimension (i.e., the rightmost specified dimension in the identifier) can be specified with slice syntax. See 5.4 for details on legal array indexing in SystemVerilog.

The *start_addr* and *finish_addr* arguments apply to the addresses of the unpacked array selected by *memory_name*. This address range represents the highest dimension of data in the *file_name*.

When slice syntax is used in the *memory_name* argument, any *start_addr* and *finish_addr* arguments must fall within the bounds of the slice's range.

The direction of the highest dimension's file entries is given by the relative magnitudes of *start_addr* and *finish_addr*, as is the case in Verilog.

23. Compiler directives

23.1 Introduction

NOTE—Verilog provides the `'define` text substitution macro compiler directive. A macro can contain arguments, whose values can be set for each instance of the macro. For example:

```
'define NAND(dval) nand #(dval)

'NAND(3)        i1 (y, a, b); //'NAND(3) macro substitutes with: nand #(3)

'NAND(3:4:5)    i2 (o, c, d); //'NAND(3:4:5) macro substitutes with: nand #(3:4:5)
```

SystemVerilog enhances the capabilities of the `'define` compiler directive to support the construction of string literals and identifiers.

Verilog provides the `` `include `` file inclusion compiler directive. SystemVerilog enhances the capabilities to support standard include specification and enhances the `` `include `` directive to accept a file name constructed with a macro.

23.2 'define macros

In Verilog, the `'define` macro text can include a backslash (\ ) at the end of a line to show continuation on the next line.

In SystemVerilog, the macro text can also include `` `" ``, `` `\`" ``, and `` `` ``.

An `` `" `` overrides the usual lexical meaning of " and indicates that the expansion should include an actual quotation mark. This allows string literals to be constructed from macro arguments.

A `` `\`" `` indicates that the expansion should include the escape sequence \". For example:

```
`define msg(x,y) `"x: `\`"y`\`"`"
```

This expands

```
$display(`msg(left side,right side));
```

to

```
$display("left side: \"right side\"");
```

A `` `` `` delimits lexical tokens without introducing white space, allowing identifiers to be constructed from arguments. For example:

```
`define foo(f) f``_suffix
```

This expands

```
'foo(bar)
```

to

```
bar_suffix
```

The `` `include `` directive can be followed by a macro, instead of a literal string:

```
`define home(filename) `"/home/foo/filename`"
`include `home(myfile)
```

23.3 `include

The syntax of the `include compiler directive is as follows:

```
include_compiler_directive ::=
    `include "filename"
  | `include <filename>
```

When the filename is an absolute path, only that filename is included and only the double quote form of the `include can be used.

When the double quote ("filename") version is used, the behavior of `include is unchanged from Verilog.

When the angle bracket (<filename>) notation is used, then only the vendor-defined location containing files defined by the language standard is searched. Relative path names given inside the < > are interpreted relative to the vendor-defined location in all cases.

23.4 `begin_keywords and `end_keywords

SystemVerilog extends the **`begin_keywords** and **`end_keywords** defined in IEEE Std 1364 by adding an "1800-2005" *version_specifier*. The *version_specifier* specifies the valid set of reserved keywords in effect when a design unit is parsed by an implementation. The `begin_keywords and `end_keywords directives can only be specified outside of a design element (**module**, **primitive**, **configuration**, **interface**, **program**, or **package**). The `begin_keywords directive affects all source code that follows the directive, even across source code file boundaries, until the matching `end_keywords directive is encountered.

The *version_specifier* "1800-2005" specifies that only the identifiers listed as reserved keywords in the IEEE Std 1800-2005 are considered to be reserved words. These identifiers are listed in Table 23-1. The `begin_keywords and `end_keywords directives only specify the set of identifiers that are reserved as keywords. The directives do not affect the semantics, tokens, and other aspects of the SystemVerilog Verilog language.

Table 23-1—IEEE Std 1800-2005 reserved keywords

alias	endmodule	matches	small
always	endpackage	medium	solve
always_comb	endprimitive	modport	specify
always_ff	endprogram	module	specparam
always_latch	endproperty	nand	static
and	endspecify	negedge	string
assert	endsequence	new	strong0
assign	endtable	nmos	strong1
assume	endtask	nor	struct
automatic	enum	noshowcancelled	super
before	event	not	supply0
begin	expect	notif0	supply1
bind	export	notif1	table
bins	extends	null	tagged
binsof	extern	or	task
bit	final	output	this
break	first_match	package	throughout
buf	for	packed	time
bufif0	force	parameter	timeprecision
bufif1	foreach	pmos	timeunit
byte	forever	posedge	tran
case	fork	primitive	tranif0
casex	forkjoin	priority	tranif1
casez	function	program	tri
cell	generate	property	tri0
chandle	genvar	protected	tri1
class	highz0	pull0	triand
clocking	highz1	pull1	trior
cmos	if	pulldown	trireg
config	iff	pullup	type
const	ifnone	pulsestyle_onevent	typedef
constraint	ignore_bins	pulsestyle_ondetect	union
context	illegal_bins	pure	unique
continue	import	rand	unsigned
cover	incdir	randc	use
covergroup	include	randcase	uwire
coverpoint	initial	randsequence	var
cross	inout	rcmos	vectored
deassign	input	real	virtual
default	inside	realtime	void
defparam	instance	ref	wait
design	int	reg	wait_order
disable	integer	release	wand
dist	interface	repeat	weak0
do	intersect	return	weak1
edge	join	rnmos	while
else	join_any	rpmos	wildcard
end	join_none	rtran	wire
endcase	large	rtranif0	with
endclass	liblist	rtranif1	within
endclocking	library	scalared	wor
endconfig	local	sequence	xnor
endfunction	localparam	shortint	xor
endgenerate	logic	shortreal	
endgroup	longint	showcancelled	
endinterface	macromodule	signed	

In the example below, it is assumed that the definition of module m1 does not have a `begin_keywords directive specified prior to the module definition. Without this directive, the set of reserved keywords in effect for this module shall be the implementation's default set of reserved keywords.

```
module m1;  // module definition with no `begin_keywords directive
  ...
endmodule
```

The following example specifies a `begin_keywords "1364-2001" directive. The source code within the module uses the identifier logic as a variable name. The `begin_keywords directive would be necessary in this example if an implementation uses IEEE Std 1800-2005 as its default set of keywords because logic is a reserved keyword in SystemVerilog. Specifying that the "1364-1995" or "1364-2005" Verilog keyword lists should be used would also work with this example.

```
`begin_keywords "1364-2001"   // use IEEE Std 1364-2001 Verilog keywords
module m2 (...);
  reg [63:0] logic;           // OK: "logic" is not a keyword in 1364-2001
  ...
endmodule
`end_keywords
```

The next example is the same code as the previous example, except that it explicitly specifies that the IEEE Std 1800-2005 SystemVerilog keywords should be used. This example shall result in an error because logic is reserved as a keyword in this standard.

```
`begin_keywords "1800-2005"   // use IEEE Std 1800-2005 SystemVerilog keywords
module m2 (...);
  reg [63:0] logic;           // ERROR: "logic" is a keyword in 1800-2005
  ...
endmodule
`end_keywords
```

The example below specifies a `begin_keywords directive on an **interface** declaration. The directive specifies that an implementation shall use the set of reserved keywords specified in this standard.

```
`begin_keywords "1800-2005"   // use IEEE Std 1800-2005 SystemVerilog keywords
interface if1 (...);
  ...
endinterface
`end_keywords
```

The next example is nearly identical to the one above, except that the `begin_keywords directive specifies that the IEEE 1364 Verilog set of keywords are to be used. This example shall result in errors because the identifiers interface and endinterface are not reserved keywords in IEEE Std 1364.

```
`begin_keywords "1364-2005"  // use IEEE 1364 Verilog keywords
interface if2 (...);       // ERROR: "interface" is not a keyword in 1364-2005
  ...
endinterface               // ERROR: "endinterface" is not a keyword in 1364-2005
`end_keywords
```

24. Value change dump (VCD) data

24.1 Introduction

NOTE—SystemVerilog extends the Verilog VCD file format to support certain SystemVerilog data types.

24.2 VCD extensions

SystemVerilog does not extend the VCD format. Some SystemVerilog types can be dumped into a standard VCD file by masquerading as a Verilog type. Table 24-1 lists the basic SystemVerilog types and their mapping to a Verilog type for VCD dumping.

Table 24-1—VCD type mapping

SystemVerilog	Verilog	Size
`bit`	`reg`	Total size of packed dimension
`logic`	`reg`	Total size of packed dimension
`int`	`integer`	32
`shortint`	**`reg`**	16
`longint`	**`reg`**	64
`byte`	`reg`	8
`enum`	`integer`	32
`shortreal`	**`real`**	—

Packed arrays and structures are dumped as a single vector of **`reg`**. Multiple packed array dimensions are collapsed into a single dimension.

If an **`enum`** declaration specified a type, it is dumped as that type rather than the default shown above.

Unpacked structures appear as named **`fork`**...**`join`** blocks, and their member elements of the structure appear as the types above. Because named **`fork`**...**`join`** blocks with variable declarations are seldom used in testbenches and hardware models, this makes structures easy to distinguish from variables declared in **`begin`**...**`end`** blocks, which are more frequently used in testbenches and models.

As in Verilog, unpacked arrays and automatic variables are not dumped.

NOTE—The current VCD format does not indicate whether a variable has been declared as **`signed`** or **`unsigned`**.

25. Deprecated constructs

25.1 Introduction

NOTE—Certain Verilog language features can be simulation inefficient, easily abused, and the source of design problems. These features are being considered for removal from the SystemVerilog language if there is an alternate method for these features.

The Verilog language features that have been identified in this standard as ones that can be removed from Verilog are **defparam** and procedural **assign**/**deassign**.

25.2 Defparam statements

The **defparam** method of specifying the value of a parameter can be a source of design errors and can be an impediment to tool implementation due to its usage of hierarchical paths. The **defparam** statement does not provide a capability that cannot be done by another method that avoids these problems. Therefore, the **defparam** statement is on a deprecation list. In other words, a future revision of IEEE Std 1364 might not require support for this feature. This current standard still requires tools to support the **defparam** statement. However, users are strongly encouraged to migrate their code to use one of the alternate methods of parameter redefinition.

Prior to the acceptance of Verilog-2001, it was common practice to change one or more parameters of instantiated modules using a separate **defparam** statement. The **defparam** statements can be a source of tool complexity and design problems.

A **defparam** statement can precede the instance to be modified, can follow the instance to be modified, can be at the end of the file that contains the instance to be modified, can be in a separate file from the instance to be modified, can modify parameters hierarchically that in turn must again be passed to other **defparam** statements to modify, and can modify the same parameter from two different **defparam** statements (with undefined results). Due to the many ways that a **defparam** can modify parameters, a Verilog compiler cannot ensure the final parameter values for an instance until after all of the design files are compiled.

Prior to Verilog-2001, the only other method available to change the values of parameters on instantiated modules was to use implicit in-line parameter redefinition. This method uses `#(parameter_value)` as part of the module instantiation. Implicit in-line parameter redefinition syntax requires that all parameters up to and including the parameter to be changed must be placed in the correct order and must be assigned values.

Verilog-2001 introduced explicit in-line parameter redefinition, in the form `#(.parameter_name(value))`, as part of the module instantiation. This method gives the capability to pass parameters by name in the instantiation, which supplies all of the necessary parameter information to the model in the instantiation itself.

The practice of using **defparam** statements is highly discouraged. Engineers are encouraged to take advantage of the Verilog-2001 explicit in-line parameter redefinition capability.

See 6.3 for more details on parameters.

25.3 Procedural assign and deassign statements

The procedural **assign** and **deassign** statements can be a source of design errors and can be an impediment to tool implementation. The procedural **assign** and **deassign** statements do not provide a capability that cannot be done by another method that avoids these problems. Therefore, the procedural **assign** and **deassign** statements are on a deprecation list. In other words, a future revision of IEEE Std 1364 might not

require support for these statements. This current standard still requires tools to support the procedural **assign** and **deassign** statements. However, users are strongly encouraged to migrate their code to use one of the alternate methods of procedural or continuous assignments.

Verilog has two forms of the **assign** statement:

— Continuous assignments, placed outside of any procedures
— Procedural continuous assignments, placed within a procedure

Continuous assignment statements are a separate process that are active throughout simulation. The continuous assignment statement accurately represents combinational logic at an RTL level of modeling and is frequently used.

Procedural continuous assignment statements become active when the **assign** statement is executed in the procedure. The process can be deactivated using a **deassign** statement. The procedural **assign** and **deassign** statements are seldom needed to model hardware behavior. In the unusual circumstances where the behavior of procedural continuous assignments are required, the same behavior can be modeled using the procedural **force** and **release** statements.

Allowing the **assign** statement to be used both inside and outside a procedural block causes confusion and is a source of errors in Verilog models. The practice of using the **assign** and **deassign** statements inside of procedural blocks is highly discouraged.

26. Direct programming interface (DPI)

26.1 Overview

This clause highlights the DPI and provides a detailed description of the SystemVerilog layer of the interface. The C layer is defined in Annex F.

DPI is an interface between SystemVerilog and a foreign programming language. It consists of two separate layers: the SystemVerilog layer and a foreign language layer. Both sides of DPI are fully isolated. Which programming language is actually used as the foreign language is transparent and irrelevant for the System-Verilog side of this interface. Neither the SystemVerilog compiler nor the foreign language compiler is required to analyze the source code in the other's language. Different programming languages can be used and supported with the same intact SystemVerilog layer. For now, however, SystemVerilog defines a foreign language layer only for the C programming language. See Annex F for more details.

The motivation for this interface is two-fold. The methodological requirement is that the interface should allow a heterogeneous system to be built (a design or a testbench) in which some components can be written in a language (or more languages) other than SystemVerilog, hereinafter called the *foreign language*. On the other hand, there is also a practical need for an easy and efficient way to connect existing code, usually written in C or C++, without the knowledge and the overhead of PLI or VPI.

DPI follows the principle of a black box: the specification and the implementation of a component are clearly separated, and the actual implementation is transparent to the rest of the system. Therefore, the actual programming language of the implementation is also transparent, although this standard defines only C linkage semantics. The separation between SystemVerilog code and the foreign language is based on using functions as the natural encapsulation unit in SystemVerilog. By and large, any function can be treated as a black box and implemented either in SystemVerilog or in the foreign language in a transparent way, without changing its calls.

26.1.1 Tasks and functions

DPI allows direct inter-language function calls between the languages on either side of the interface. Specifically, functions implemented in a foreign language can be called from SystemVerilog; such functions are referred to as *imported functions*. SystemVerilog functions that are to be called from a foreign code shall be specified in export declarations (see 26.6 for more details). DPI allows for passing SystemVerilog data between the two domains through function arguments and results. There is no intrinsic overhead in this interface.

It is also possible to perform task enables across the language boundary. Foreign code can call SystemVerilog tasks, and native Verilog code can call imported tasks. An imported task has the same semantics as a native Verilog task: it never returns a value, and it can consume simulation time.

All functions used in DPI are assumed to complete their execution instantly and consume zero simulation time, just as normal SystemVerilog functions. DPI provides no means of synchronization other than by data exchange and explicit transfer of control.

Every imported task and function needs to be declared. A declaration of an imported task or function is referred to as an *import declaration*. Import declarations are very similar to SystemVerilog task and function declarations. Import declarations can occur anywhere where SystemVerilog task and function definitions are permitted. An import declaration is considered to be a definition of a SystemVerilog task or function with a foreign language implementation. The same foreign task or function can be used to implement multiple SystemVerilog tasks and functions (this can be a useful way of providing differing default argument values for the same basic task or function), but a given SystemVerilog name can only be defined once per scope.

Imported task and functions can have zero or more formal **input**, **output**, and **inout** arguments. Imported tasks always return a void value and thus can only be used in statement context. Imported functions can return a result or be defined as void functions.

DPI is based entirely upon SystemVerilog constructs. The usage of imported functions is identical to the usage of native SystemVerilog functions. With few exceptions, imported functions and native functions are mutually exchangeable. Calls of imported functions are indistinguishable from calls of SystemVerilog functions. This facilitates ease of use and minimizes the learning curve. Similar interchangeable semantics exist between native SystemVerilog tasks and imported tasks.

26.1.2 Data types

SystemVerilog data types are the sole data types that can cross the boundary between SystemVerilog and a foreign language in either direction (i.e., when an imported function is called from SystemVerilog code or an exported SystemVerilog function is called from a foreign code). It is not possible to import the data types or directly use the type syntax from another language. A rich subset of SystemVerilog data types is allowed for formal arguments of import and export functions, although with some restrictions and with some notational extensions. Function result types are restricted to small values, however (see 26.4.5).

Formal arguments of an imported function can be specified as open arrays. A formal argument is an open array when a range of one or more of its dimensions, packed or unpacked, is unspecified. An open array is like a multidimensional dynamic array formal in both packed and unpacked dimensions and is thus denoted using the same syntax as dynamic arrays, using [] to denote an open dimension. This is solely a relaxation of the argument-matching rules. An actual argument shall match the formal one regardless of the range(s) for its corresponding dimension(s), which facilitates writing generalized code that can handle System-Verilog arrays of different sizes. See 26.4.6.1.

26.1.2.1 Data representation

DPI does not add any constraints on how SystemVerilog-specific data types are actually implemented. Optimal representation can be platform dependent. The layout of 2- or 4-state packed structures and arrays is implementation and platform dependent.

The implementation (representation and layout) of 4-state values, structures, and arrays is irrelevant for SystemVerilog semantics and can only impact the foreign side of the interface.

26.2 Two layers of the DPI

DPI consists of two separate layers: the SystemVerilog layer and a foreign language layer. The SystemVerilog layer does not depend on which programming language is actually used as the foreign language. Although different programming languages can be supported and used with the intact SystemVerilog layer, SystemVerilog defines a foreign language layer only for the C programming language. Nevertheless, SystemVerilog code shall look identical and its semantics shall be unchanged for any foreign language layer. Different foreign languages can require that the SystemVerilog implementation shall use the appropriate function call protocol and argument passing and linking mechanisms. This shall be, however, transparent to SystemVerilog users. SystemVerilog requires only that its implementation shall support C protocols and linkage.

26.2.1 DPI SystemVerilog layer

The SystemVerilog side of DPI does not depend on the foreign programming language. In particular, the actual function call protocol and argument passing mechanisms used in the foreign language are transparent and irrelevant to SystemVerilog. SystemVerilog code shall look identical regardless of what code the

foreign side of the interface is using. The semantics of the SystemVerilog side of the interface is independent from the foreign side of the interface.

This clause does not constitute a complete interface specification. It only describes the functionality, semantics, and syntax of the SystemVerilog layer of the interface. The other half of the interface, the foreign language layer, defines the actual argument passing mechanism and the methods to access (read/write) formal arguments from the foreign code. See Annex F for more details.

26.2.2 DPI foreign language layer

The foreign language layer of the interface (which is transparent to SystemVerilog) shall specify how actual arguments are passed, how they can be accessed from the foreign code, how SystemVerilog-specific data types (such as **logic** and **packed**) are represented, and how they are translated to and from some predefined C-like types.

The data types allowed for formal arguments and results of imported functions or exported functions are generally SystemVerilog types (with some restrictions and with notational extensions for open arrays). Users are responsible for specifying in their foreign code the native types equivalent to the SystemVerilog types used in imported declarations or export declarations. Software tools, like a SystemVerilog compiler, can facilitate the mapping of SystemVerilog types onto foreign native types by generating the appropriate function headers.

The SystemVerilog compiler or simulator shall generate and/or use the function call protocol and argument passing mechanisms required for the intended foreign language layer. The same SystemVerilog code (compiled accordingly) shall be usable with different foreign language layers, regardless of the data access method assumed in a specific layer. Annex G defines the DPI foreign language layer for the C programming language.

26.3 Global name space of imported and exported functions

Every task or function imported to SystemVerilog must eventually resolve to a global symbol. Similarly, every task or function exported from SystemVerilog defines a global symbol. Thus the tasks and functions imported to and exported from SystemVerilog have their own global name space of linkage names, different from compilation-unit scope name space. Global names of imported and exported tasks and functions must be unique (no overloading is allowed) and shall follow C conventions for naming; specifically, such names must start with a letter or underscore, and they can be followed by alphanumeric characters or underscores. Exported and imported tasks and functions, however, can be declared with local SystemVerilog names. Import and export declarations allow users to specify a global name for a function in addition to its declared name. Should a global name clash with a SystemVerilog keyword or a reserved name, it shall take the form of an escaped identifier. The leading backslash (\ ) character and the trailing white space shall be stripped off by the SystemVerilog tool to create the linkage identifier. After this stripping, the linkage identifier so formed must comply with the normal rules for C identifier construction. If a global name is not explicitly given, it shall be the same as the SystemVerilog task or function name. For example:

```
export "DPI" foo_plus = function \foo+ ; // "foo+" exported as "foo_plus"
export "DPI" function foo; // "foo" exported under its own name
import "DPI" init_1 = function void \init[1] (); // "init_1" is a linkage name
import "DPI" \begin = function void \init[2] (); // "begin" is a linkage name
```

The same global task or function can be referred to in multiple import declarations in different scopes or/and with different SystemVerilog names (see 26.4.4).

Multiple export declarations are allowed with the same *c_identifier*, explicit or implicit, as long as they are in different scopes and have the equivalent type signature (as defined in 26.4.4 for imported tasks and functions). Multiple export declarations with the same *c_identifier* in the same scope are forbidden.

It is possible to use the deprecated `"DPI"` version string syntax in an import or export declaration. This syntax indicates that the SystemVerilog 2-state and 4-state packed array argument passing convention is to be used (see F.12). In such cases, all declarations using the same *c_identifier* shall be declared with the same DPI version string syntax.

26.4 Imported tasks and functions

The usage of imported functions is similar as for native SystemVerilog functions.

26.4.1 Required properties of imported tasks and functions—semantic constraints

This subclause defines the semantic constraints imposed on imported tasks or functions. Some semantic restrictions are shared by all imported tasks or functions. Other restrictions depend on whether the special properties **pure** (see 26.4.2) or **context** (see 26.4.3) are specified for an imported task or function. A SystemVerilog compiler is not able to verify that those restrictions are observed; and if those restrictions are not satisfied, the effects of such imported task or function calls can be unpredictable.

26.4.1.1 Instant completion of imported functions

Imported functions shall complete their execution instantly and consume zero simulation time, similarly to native functions.

NOTE—Imported tasks can consume time, similar to native SystemVerilog tasks.

26.4.1.2 input, output, and inout arguments

Imported functions can have **input**, **output**, and **inout** arguments. The formal **input** arguments shall not be modified. If such arguments are changed within a function, the changes shall not be visible outside the function; the actual arguments shall not be changed.

The imported function shall not assume anything about the initial values of formal **output** arguments. The initial values of **output** arguments are undetermined and implementation dependent.

The imported function can access the initial value of a formal **inout** argument. Changes that the imported function makes to a formal **inout** argument shall be visible outside the function.

26.4.1.3 Special properties pure and context

Special properties can be specified for an imported task or function as **pure** or as **context** (see also 26.4.2 or 26.4.3).

A function whose result depends solely on the values of its input arguments and with no side effects can be specified as **pure**. This can usually allow for more optimizations and thus can result in improved simulation performance. Subclause 26.4.2 details the rules that must be obeyed by pure functions. An imported task can never be declared pure.

An imported task or function that is intended to call exported tasks or functions or to access SystemVerilog data objects other than its actual arguments (e.g., via VPI or PLI calls) must be specified as **context**. Calls of context tasks and functions are specially instrumented and can impair SystemVerilog compiler optimizations; therefore, simulation performance can decrease if the **context** property is specified when not

necessary. A task or function not specified as **context** shall not read or write any data objects from System-Verilog other then its actual arguments. For tasks or functions not specified as **context**, the effects of calling PLI, VPI, or exported SystemVerilog tasks or functions can be unpredictable and can lead to unexpected behavior; such calls can even crash. Subclause 26.4.3 details the restrictions that must be obeyed by noncontext tasks or functions.

If neither the pure nor the context attribute is used on an imported task or function, the task or function shall not access SystemVerilog data objects; however, it can perform side effects such as writing to a file or manipulating a global variable.

26.4.1.4 Memory management

The memory spaces owned and allocated by the foreign code and SystemVerilog code are disjoined. Each side is responsible for its own allocated memory. Specifically, an imported function shall not free the memory allocated by SystemVerilog code (or the SystemVerilog compiler) nor expect SystemVerilog code to free the memory allocated by the foreign code (or the foreign compiler). This does not exclude scenarios where foreign code allocates a block of memory and then passes a handle (i.e., a pointer) to that block to SystemVerilog code, which in turn calls an imported function (e.g., C standard function `free`) that directly or indirectly frees that block.

NOTE—In this last scenario, a block of memory is allocated and freed in the foreign code, even when the standard functions `malloc` and `free` are called directly from SystemVerilog code.

26.4.1.5 Reentrancy of imported tasks

A call to an imported task can result in the suspension of the currently executing thread. This occurs when an imported task calls an exported task, and the exported task executes a delay control, event control, or wait statement. Thus it is possible for an imported task's C code to be simultaneously active in multiple execution threads. Standard reentrancy considerations must be made by the C programmer. Some examples of such considerations include safely using static variables and ensuring that only thread-safe C standard library calls (MT safe) are used.

26.4.1.6 C++ exceptions

It is possible to implement DPI imported tasks and functions using C++, as long as C linkage conventions are observed at the language boundary. If C++ is used, exceptions must not propagate out of any imported task or function. Undefined behavior will result if an exception crosses the language boundary from C++ into SystemVerilog.

26.4.2 Pure functions

A **pure** function call can be safely eliminated if its result is not needed or if the previous result for the same values of input arguments is available somehow and can be reused without needing to recalculate. Only non-void functions with no **output** or **inout** arguments can be specified as **pure**. Functions specified as **pure** shall have no side effects whatsoever; their results need to depend solely on the values of their input arguments. Calls to such functions can be removed by SystemVerilog compiler optimizations or replaced with the values previously computed for the same values of the input arguments.

Specifically, a pure function is assumed not to directly or indirectly (i.e., by calling other functions) perform the following:

— Perform any file operations.
— Read or write anything in the broadest possible meaning, including input/output, environment variables, objects from the operating system or from the program or other processes, shared memory, sockets, etc.

— Access any persistent data, like global or static variables.

If a pure function does not obey the above restrictions, SystemVerilog compiler optimizations can lead to unexpected behavior, due to eliminated calls or incorrect results being used.

26.4.3 Context tasks and functions

Some DPI imported tasks or functions require that the context of their call be known. It takes special instrumentation of their call instances to provide such context; for example, an internal variable referring to the "current instance" might need to be set. To avoid any unnecessary overhead, imported task or function calls in SystemVerilog code are not instrumented unless the imported task or function is specified as **context**.

The SystemVerilog context of DPI export tasks and functions must be known when they are called, including when they are called by imports. When an import invokes the `svSetScope` utility prior to calling the export, it sets the context explicitly. Otherwise, the context will be the context of the instantiated scope where the import declaration is located. Because imports with diverse instantiated scopes can export the same task or function, multiple instances of such an export can exist after elaboration. Prior to any invocations of `svSetScope`, these export instances would have different contexts, which would reflect their imported caller's instantiated scope.

For the sake of simulation performance, an imported task or function call shall not block SystemVerilog compiler optimizations. An imported task or function not specified as **context** shall not access any data objects from SystemVerilog other than its actual arguments. Only the actual arguments can be affected (read or written) by its call. Therefore, a call of a noncontext task or function is not a barrier for optimizations. A context imported task or function, however, can access (read or write) any SystemVerilog data objects by calling PLI/VPI or by calling an export task or function. Therefore, a call to a context task or function is a barrier for SystemVerilog compiler optimizations.

Only calls of context imported tasks or functions are properly instrumented and cause conservative optimizations; therefore, only those tasks or functions can safely call all tasks or functions from other APIs, including PLI and VPI functions or exported SystemVerilog tasks or functions. For imported tasks or functions not specified as **context**, the effects of calling PLI or VPI functions or SystemVerilog tasks or functions can be unpredictable; and such calls can crash if the callee requires a context that has not been properly set. However, declaring an import context task or function does not automatically make any other simulator interface automatically available. For VPI access (or any other interface access) to be possible, the appropriate implementation-defined mechanism must still be used to enable these interface(s). Realize also that DPI calls do not automatically create or provide any handles or any special environment that can be needed by those other interfaces. It is the user's responsibility to create, manage, or otherwise manipulate the required handles or environment(s) needed by the other interfaces.

Context imported tasks or functions are always implicitly supplied a scope representing the fully qualified instance name within which the import declaration was present. This scope defines which exported SystemVerilog tasks or functions can be called directly from the imported task or function; only tasks or functions defined and exported from the same scope as the import can be called directly. To call any other exported SystemVerilog tasks or functions, the imported task or function shall first have to modify its current scope, in essence performing the foreign language equivalent of a SystemVerilog hierarchical task or function call.

Special DPI utility functions exist that allow imported task or functions to retrieve and operate on their scope. See Annex F for more details.

26.4.4 Import declarations

Each imported task or function shall be declared. Such declaration are referred to as *import declarations*. The syntax of an **import** declaration is similar to the syntax of SystemVerilog task or function prototypes (see 12.5).

Imported tasks or functions are similar to SystemVerilog tasks or functions. Imported tasks or functions can have zero or more formal **input**, **output**, and **inout** arguments. Imported functions can return a result or be defined as void functions. Imported tasks always return an **int** result as part of the DPI disable protocol and, thus, are declared in foreign code as **int** functions (see 26.7 and 26.8).

---

dpi_import_export ::=   *// from A.2.6*
      import dpi_spec_string [dpi_function_import_property] [c_identifier **=**] dpi_function_proto **;**
    | **import** dpi_spec_string [dpi_task_import_property] [c_identifier **=**] dpi_task_proto **;**
    | **export** dpi_spec_string [c_identifier **=**] **function** function_identifier **;**
    | **export** dpi_spec_string [c_identifier =] **task** task_identifier **;**

dpi_spec_string ::= **"DPI-C"** | **"DPI"**

dpi_function_import_property ::= **context** | **pure**

dpi_task_import_property ::= **context**

dpi_function_proto[8,9] ::= function_prototype

dpi_task_proto[9] ::= task_prototype

function_prototype ::= **function** function_data_type function_identifier **(** [tf_port_list] **)**

task_prototype ::= **task** task_identifier **(** [tf_port_list] **)**   *// from A.2.7*

Details: *// from A.10*

8) dpi_function_proto return types are restricted to small values, per 26.4.5.
9) Formals of dpi_function_proto and dpi_task_proto cannot use pass by reference mode and class types cannot be passed at all; for the complete set of restrictions see 26.4.6.

Syntax 26-1—DPI import declaration syntax (excerpt from Annex A)

An import declaration specifies the task or function name, function result type, and types and directions of formal arguments. It can also provide optional default values for formal arguments. Formal argument names are optional unless argument binding by name is needed. An import declaration can also specify an optional task or function property. Imported functions can have the properties context or pure; imported tasks can have the property **context**.

Because an import declaration is equivalent to defining a task or function of that name in the SystemVerilog scope in which the import declaration occurs, and thus multiple imports of the same task or function name into the same scope are forbidden.

NOTE—This declaration scope is particularly important in the case of imported context tasks or functions (see 26.4.3); for noncontext imported tasks or functions the declaration scope has no other implications other than defining the visibility of the task or function.

The *dpi_spec_string* can take values "DPI-C" and "DPI". "DPI" is used to indicate that the deprecated SystemVerilog packed array argument passing semantics is to be used. In this semantics, arguments are passed in actual simulator representation format rather than in canonical format, as is the case with "DPI-C".

Use of the string "DPI" shall generate a compile-time error. The error message shall contain the following information:

— "DPI" is deprecated and should be replaced with "DPI-C".

— Use of the "DPI-C" string may require changes in the DPI application's C code.

For more information on using deprecated "DPI" access to packed data, see F.12.

The *c_identifier* provides the linkage name for this task or function in the foreign language. If not provided, this defaults to the same identifier as the SystemVerilog task or function name. In either case, this linkage name must conform to C identifier syntax. An error shall occur if the *c_identifier*, either directly or indirectly, does not conform to these rules.

For any given *c_identifier* (whether explicitly defined with *c_identifier=* or automatically determined from the task or function name), all declarations, regardless of scope, must have exactly the same type signature. The signature includes the return type and the number, order, direction, and types of each and every argument. The type includes dimensions and bounds of any arrays or array dimensions. The signature also includes the **pure**/**context** qualifiers that can be associated with an extern definition, and it includes the value of the *dpi_spec_string*.

It is permitted to have multiple declarations of the same imported or exported task or function in different scopes; therefore, argument names and default values can vary, provided the type compatibility constraints are met.

A formal argument name is required to separate the packed and the unpacked dimensions of an array.

The qualifier **ref** cannot be used in import declarations. The actual implementation of argument passing depends solely on the foreign language layer and its implementation and shall be transparent to the System-Verilog side of the interface.

The following are examples of external declarations.

```
import "DPI" function void myInit();

// from standard math library
import "DPI" pure function real sin(real);

// from standard C library: memory management
import "DPI" function chandle malloc(int size); // standard C function
import "DPI" function void free(chandle ptr); // standard C function

// abstract data structure: queue
import "DPI" function chandle newQueue(input string name_of_queue);

// Note the following import uses the same foreign function for
// implementation as the prior import, but has different SystemVerilog name
// and provides a default value for the argument.
import "DPI" newQueue=function chandle newAnonQueue(input string s=null);
import "DPI" function chandle newElem(bit [15:0]);
import "DPI" function void enqueue(chandle queue, chandle elem);
import "DPI" function chandle dequeue(chandle queue);

// miscellanea
import "DPI" function bit [15:0] getStimulus();
import "DPI" context function void processTransaction(chandle elem,
                                    output logic [64:1] arr [0:63]);
import "DPI" task checkResults(input string s, bit [511:0] packet);
```

26.4.5 Function result

An imported function declaration must explicitly specify a data type or void for the type of the function's return result. Function result types are restricted to small values. The following SystemVerilog data types are allowed for imported function results:

— **void**, **byte**, **shortint**, **int**, **longint**, **real**, **shortreal**, **chandle**, and **string**
— Scalar values of type **bit** and **logic**

The same restrictions apply for the result types of exported functions.

26.4.6 Types of formal arguments

A rich subset of SystemVerilog data types is allowed for formal arguments of import and export tasks or functions. Generally, C-compatible types, packed types, and user-defined types built of types from these two categories can be used for formal arguments of DPI tasks or functions. The set of permitted types is defined inductively.

The following SystemVerilog types are the only permitted types for formal arguments of import and export tasks or functions:

— **void**, **byte**, **shortint**, **int**, **longint**, **real**, **shortreal**, **chandle**, and **string**
— Scalar values of type **bit** and **logic**
— Packed arrays, structs, and unions composed of types **bit** and **logic**. Every packed type is eventually equivalent to a packed one-dimensional array. On the foreign language side of the DPI, all packed types are perceived as packed one-dimensional arrays regardless of their declaration in the SystemVerilog code.
— Enumeration types interpreted as the type associated with that enumeration
— Types constructed from the supported types with the help of the constructs:

    — **struct**

    — **union**

    — Unpacked array

    — **typedef**

The following caveats apply for the types permitted in DPI:

— Enumerated data types are not supported directly. Instead, an enumerated data type is interpreted as the type associated with that enumerated type.
— SystemVerilog does not specify the actual memory representation of packed structures or any arrays, packed or unpacked. Unpacked structures have an implementation-dependent packing, normally matching the C compiler.
— In exported DPI tasks or functions, it is erroneous to declare formal arguments of dynamic array types.
— The actual memory representation of SystemVerilog data types is transparent for SystemVerilog semantics and irrelevant for SystemVerilog code. It can be relevant for the foreign language code on the other side of the interface, however; a particular representation of the SystemVerilog data types can be assumed. This shall not restrict the types of formal arguments of imported tasks or functions, with the exception of unpacked arrays. SystemVerilog implementation can restrict which SystemVerilog unpacked arrays are passed as actual arguments for a formal argument that is a sized array, although they can be always passed for an unsized (i.e., open) array. Therefore, the correctness of an actual argument might be implementation dependent. Nevertheless, an open array provides an implementation-independent solution.

26.4.6.1 Open arrays

The size of the packed dimension, the unpacked dimension, or both dimensions can remain unspecified; such cases are referred to as *open arrays* (or *unsized arrays*). Open arrays allow the use of generic code to handle different sizes.

Formal arguments of imported functions can be specified as open arrays. (Exported SystemVerilog functions cannot have formal arguments specified as open arrays.) A formal argument is an open array when a range of one or more of its dimensions is unspecified (denoted by using square brackets, []). This is solely a relaxation of the argument-matching rules. An actual argument shall match the formal one regardless of the range(s) for its corresponding dimension(s), which facilitates writing generalized code that can handle SystemVerilog arrays of different sizes.

Although the packed part of an array can have an arbitrary number of dimensions, in the case of open arrays only a single dimension is allowed for the packed part. This is not very restrictive, however, because any packed type is eventually equivalent to a one-dimensional packed array. The number of unpacked dimensions is not restricted.

If a formal argument is specified as an open array with a range of its packed or one or more of its unpacked dimensions unspecified, then the actual argument shall match the formal one, regardless of its dimensions and sizes of its linearized packed or unpacked dimensions corresponding to an unspecified range of the formal argument, respectively.

Examples of types of formal arguments (empty square brackets [] denote open array):

```
logic
bit [8:1]
bit[]
bit [7:0] array8x10 [1:10] // array8x10 is a formal arg name
logic [31:0] array32xN []  // array32xN is a formal arg name
logic [] arrayNx3 [3:1]    // arrayNx3 is a formal arg name
bit [] arrayNxN []         // arrayNxN is a formal arg name
```

Example of complete import declarations:

```
import "DPI" function void foo(input logic [127:0]);
import "DPI" function void boo(logic [127:0] i []); // open array of 128-bit
```

Example of the use of open arrays for different sizes of actual arguments:

```
typedef struct {int i; ... } MyType;

import "DPI" function void foo(input MyType i [][]);
      /* 2-dimensional unsized unpacked array of MyType */

MyType a_10x5 [11:20][6:2];
MyType a_64x8 [64:1][-1:-8];

foo(a_10x5);
foo(a_64x8);
```

26.5 Calling imported functions

The usage of imported functions is identical to the usage of native SystemVerilog functions. Hence the usage and syntax for calling imported functions is identical to the usage and syntax of native SystemVerilog

functions. Specifically, arguments with default values can be omitted from the call; arguments can be bound by name if all formal arguments are named.

26.5.1 Argument passing

Argument passing for imported functions is ruled by the WYSIWYG principle: What You Specify Is What You Get (see 26.5.1.1). The evaluation order of formal arguments follows general SystemVerilog rules.

Argument compatibility and coercion rules are the same as for native SystemVerilog functions. If a coercion is needed, a temporary variable is created and passed as the actual argument. For **input** and **inout** arguments, the temporary variable is initialized with the value of the actual argument with the appropriate coercion. For **output** or **inout** arguments, the value of the temporary variable is assigned to the actual argument with the appropriate conversion. The assignments between a temporary and the actual argument follow general SystemVerilog rules for assignments and automatic coercion.

On the SystemVerilog side of the interface, the values of actual arguments for formal input arguments of imported functions shall not be affected by the callee. The initial values of formal output arguments of imported functions are unspecified (and can be implementation dependent), and the necessary coercions, if any, are applied as for assignments. Imported functions shall not modify the values of their input arguments.

For the SystemVerilog side of the interface, the semantics of arguments passing is as if **input** arguments are passed by *copy-in*, **output** arguments are passed by *copy-out*, and **inout** arguments were passed by *copy-in, copy-out*. The terms *copy-in* and *copy-out* do not impose the actual implementation; they refer only to "hypothetical assignment".

The actual implementation of argument passing is transparent to the SystemVerilog side of the interface. In particular, it is transparent to SystemVerilog whether an argument is actually passed by value or by reference. The actual argument passing mechanism is defined in the foreign language layer. See Annex F for more details.

26.5.1.1 WYSIWYG principle

The WYSIWYG principle guarantees the types of formal arguments of imported functions: an actual argument is guaranteed to be of the type specified for the formal argument, with the exception of open arrays (for which unspecified ranges are statically unknown). Formal arguments, other than open arrays, are fully defined by import declaration; they shall have ranges of packed or unpacked arrays exactly as specified in the import declaration. Only the declaration site of the imported function is relevant for such formal arguments.

Another way to state this is that no compiler (either C or SystemVerilog) can make argument coercions between a caller's declared formal and the callee's declared formals. This is because the callee's formal arguments are declared in a different language from the caller's formal arguments; hence there is no visible relationship between the two sets of formals. Users are expected to understand all argument relationships and provide properly matched types on both sides of the interface.

Formal arguments defined as open arrays have the size and ranges of the corresponding actual arguments, i.e., have the ranges of packed or unpacked arrays exactly as that of the actual argument. The unsized ranges of open arrays are determined at a call site; the rest of the type information is specified at the import declaration.

Therefore, if a formal argument is declared as **bit** [15:8] b [], then the import declaration specifies that the formal argument is an unpacked array of packed bit array with bounds 15 to 8, while the actual argument used at a particular call site defines the bounds for the unpacked part for that call.

It is sometimes permissible to pass a dynamic array as an actual argument to an imported DPI task or function. The rules for passing dynamic array actual arguments to imported DPI tasks and functions are identical to the rules for native SystemVerilog tasks and functions. Refer to 5.8 for details on such use of dynamic arrays.

26.5.2 Value changes for output and inout arguments

The SystemVerilog simulator is responsible for handling value changes for **output** and **inout** arguments. Such changes shall be detected and handled after control returns from imported functions to SystemVerilog code.

For **output** and **inout** arguments, the value propagation (i.e., value change events) happens as if an actual argument was assigned a formal argument immediately after control returns from imported functions. If there is more than one argument, the order of such assignments and the related value change propagation follows general SystemVerilog rules.

26.6 Exported functions

DPI allows calling SystemVerilog functions from another language. However, such functions must adhere to the same restrictions on argument types and results as imposed on imported functions. It is an error to export a function that does not satisfy such constraints.

SystemVerilog functions that can be called from foreign code need to be specified in **export** declarations. Export declarations are allowed to occur only in the scope in which the function being exported is defined. Only one export declaration per function is allowed in a given scope.

One important restriction exists. Class member functions cannot be exported, but all other SystemVerilog functions can be exported.

Similar to import declarations, **export** declarations can define an optional *c_identifier* to be used in the foreign language when calling an exported function.

```
dpi_import_export ::=                                                          // from A.2.6
    | export dpi_spec_string [ c_identifier = ] function function_identifier ;
dpi_spec_string ::= "DPI-C" | "DPI"
```

Syntax 26-2—DPI export declaration syntax (excerpt from Annex A)

The *c_identifier* is optional here. It defaults to *function_identifier*. For rules describing *c_identifier*, see 26.3. No two functions in the same SystemVerilog scope can be exported with the same explicit or implicit *c_identifier*. The export declaration and the definition of the corresponding SystemVerilog function can occur in any order. Only one export declaration is permitted per SystemVerilog function, and all export functions are always context functions.

26.7 Exported tasks

SystemVerilog allows tasks to be called from a foreign language, similar to functions. Such tasks are termed *exported tasks*.

All aspects of exported functions described above in 26.6 apply to exported tasks. This includes legal declaration scopes as well as usage of the optional *c_identifier*.

It is never legal to call an exported task from within an imported function. This semantics is identical to native SystemVerilog semantics, in which it is illegal for a function to perform a task enable.

It is legal for an imported task to call an exported task only if the imported task is declared with the `context` property. See 26.4.3 for more details.

One difference between exported tasks and exported functions is that SystemVerilog tasks do not have return value types. The return value of an exported task is an `int` value that indicates if a disable is active or not on the current execution thread.

Similarly, imported tasks return an `int` value that is used to indicate that the imported task has acknowledged a disable. See 26.8 for more detail on disables in DPI.

26.8 Disabling DPI tasks and functions

It is possible for a `disable` statement to disable a block that is currently executing a mixed language call chain. When a DPI import task or function is disabled, the C code is required to follow a simple disable protocol. The protocol gives the C code the opportunity to perform any necessary resource cleanup, such as closing open file handles, closing open VPI handles, or freeing heap memory.

An imported task or function is said to be in the disabled state when a `disable` statement somewhere in the design targets either it or a parent for disabling. An imported task or function can only enter the disabled state immediately after the return of a call to an exported task or function. An important aspect of the protocol is that disabled import tasks and functions must programmatically acknowledge that they have been disabled. A task or function can determine that it is in the disabled state by calling the API function `svIsDisabledState()`.

The protocol is composed of the following items:

a)   When an exported task returns due to a disable, it must return a value of 1. Otherwise, it must return 0.

b)   When an imported task returns due to a disable, it must return a value of 1. Otherwise, it must return 0.

c)   Before an imported function returns due to a disable, it must call the API function `svAckDisabledState()`.

d)   Once an imported task or function enters the disabled state, it is illegal for the current function invocation to make any further calls to exported tasks or functions.

Item b, item c, and item d are mandatory behavior for imported DPI tasks and functions. It is the responsibility of the DPI programmer to correctly implement the behavior.

Item a is guaranteed by SystemVerilog simulators. In addition, simulators must implement checks to ensure that item b, item c, and item d are correctly followed by imported tasks and functions. If any protocol item is not correctly followed, a fatal simulation error is issued.

If an exported task itself is the target of a disable, its parent imported task is not considered to be in the disabled state when the exported task returns. In such cases, the exported task shall return value 0, and calls to `svIsDisabledState()` shall return 0 as well.

When a DPI imported task or function returns due to a disable, the values of its `output` and `inout` parameters are undefined. Similarly, function return values are undefined when an imported function returns due to a disable. C programmers can return values from disabled functions, and C programmers can write values into the locations of `output` and `inout` parameters of imported tasks or functions. However,

SystemVerilog simulators are not obligated to propagate any such values to the calling SystemVerilog code if a disable is in effect.

27. SystemVerilog VPI object model

27.1 Introduction

NOTE—SystemVerilog extends the Verilog procedural interface (VPI) object diagrams to support SystemVerilog constructs. The VPI object diagrams document the properties of objects and the relationships of objects. How these diagrams illustrate this information is explained in Clause 26 of IEEE Std 1364. The SystemVerilog extensions to the VPI diagrams are in the form of changes to or additions to the diagrams contained in IEEE Std 1364.

Table 27-1 summarizes the changes and additions made to the Verilog VPI object diagrams.

Table 27-1—Verilog VPI object diagram changes and additions

Diagram	Notes
27.2 Module	Supersedes IEEE Std 1364, 26.6.1
27.3 Interface	Addition to IEEE 1364 VPI diagrams
27.4 Modport	Addition to IEEE 1364 VPI diagrams
27.5 Interface task and function declaration	Addition to IEEE 1364 VPI diagrams
27.6 Program	Addition to IEEE 1364 VPI diagrams
27.7 Instance	Addition to IEEE 1364 VPI diagrams
27.8 Instance arrays	Supersedes IEEE Std 1364, 26.6.2
27.9 Scope	Supersedes IEEE Std 1364, 26.6.3
27.10 IO declaration	Supersedes IEEE Std 1364, 26.6.4
27.11 Ports	Supersedes IEEE Std 1364, 26.6.5
27.12 Reference objects	Addition to IEEE 1364 VPI diagrams
27.13 Nets	Supersedes IEEE Std 1364, 26.6.6
27.14 Variables	Supersedes IEEE Std 1364, 26.6.7, 26.6.8
27.15 Variable select	Addition to IEEE 1364 VPI diagrams
27.16 Variable drivers and loads	Supersedes IEEE Std 1364, 26.6.23
27.17 Typespec	Addition to IEEE 1364 VPI diagrams
27.18 Structures and unions	Addition to IEEE 1364 VPI diagrams
27.19 Named events	Supersedes IEEE Std 1364, 26.6.11
27.20 Parameter	Supersedes parameter in IEEE Std 1364, 26.6.12
27.21 Class object definition	Addition to IEEE 1364 VPI diagrams
27.22 Class variables	Addition to IEEE 1364 VPI diagrams
27.23 Constraint, constraint ordering, distribution	Addition to IEEE 1364 VPI diagrams
27.24 Constraint expression	Addition to IEEE 1364 VPI diagrams
27.25 Module path, path term	Supersedes IEEE Std 1364, 26.6.15
27.26 Task and function declaration	Supersedes IEEE Std 1364, 26.6.18
27.27 Task and function call	Supersedes IEEE Std 1364, 26.6.19

Table 27-1—Verilog VPI object diagram changes and additions *(continued)*

Diagram	Notes
27.28 Frames	Supersedes IEEE Std 1364, 26.6.20
27.29 Threads	Addition to IEEE 1364 VPI diagrams
27.30 Clocking block	Addition to IEEE 1364 VPI diagrams
27.31 Assertion	Addition to IEEE 1364 VPI diagrams
27.32 Concurrent assertions	Addition to IEEE 1364 VPI diagrams
27.33 Property declaration	Addition to IEEE 1364 VPI diagrams
27.34 Property specification	Addition to IEEE 1364 VPI diagrams
27.35 Sequence declaration	Addition to IEEE 1364 VPI diagrams
27.36 Sequence expression	Addition to IEEE 1364 VPI diagrams
27.37 Multiclock sequence expression	Addition to IEEE 1364 VPI diagrams
27.38 Simple expressions	Supersedes IEEE Std 1364, 26.6 25
27.39 Expressions	Supersedes IEEE Std 1364, 26.6.26
27.40 Atomic statement	Supersedes atomic statement in IEEE Std 1364, 26.6.27
27.41 Event statement	Supersedes event statement in IEEE Std 1364, 26.6.27
27.42 Process	Supersedes process in IEEE Std 1364, 26.6.27
27.43 Assignment	Supersedes IEEE Std 1364, 26.6.28
27.44 Event control	Supersedes IEEE Std 1364, 26.6.30
27.45 Waits	Supersedes wait object in IEEE Std 1364, 26.6.32
27.46 if, if-else	Supersedes IEEE Std 1364, 26.6.35
27.47 case, pattern	Supersedes IEEE Std 1364, 26.6.36
27.48 Expect	Addition to IEEE 1364 VPI diagrams
27.49 For	Supersedes IEEE Std 1364, 26.6.33
27.50 Do-while, foreach	Addition to IEEE 1364 VPI diagrams
27.51 Alias statement	Addition to IEEE 1364 VPI diagrams
27.52 Disable	Supersedes IEEE Std 1364, 26.6.38
27.53 Return	Addition to IEEE 1364 VPI diagrams
27.54 Attribute	Supersedes IEEE Std 1364, 26.6.42
27.55 Generates	Supersedes IEEE Std 1364, 26.6.44

27.2 Module (supersedes 26.6.1 of IEEE Std 1364)
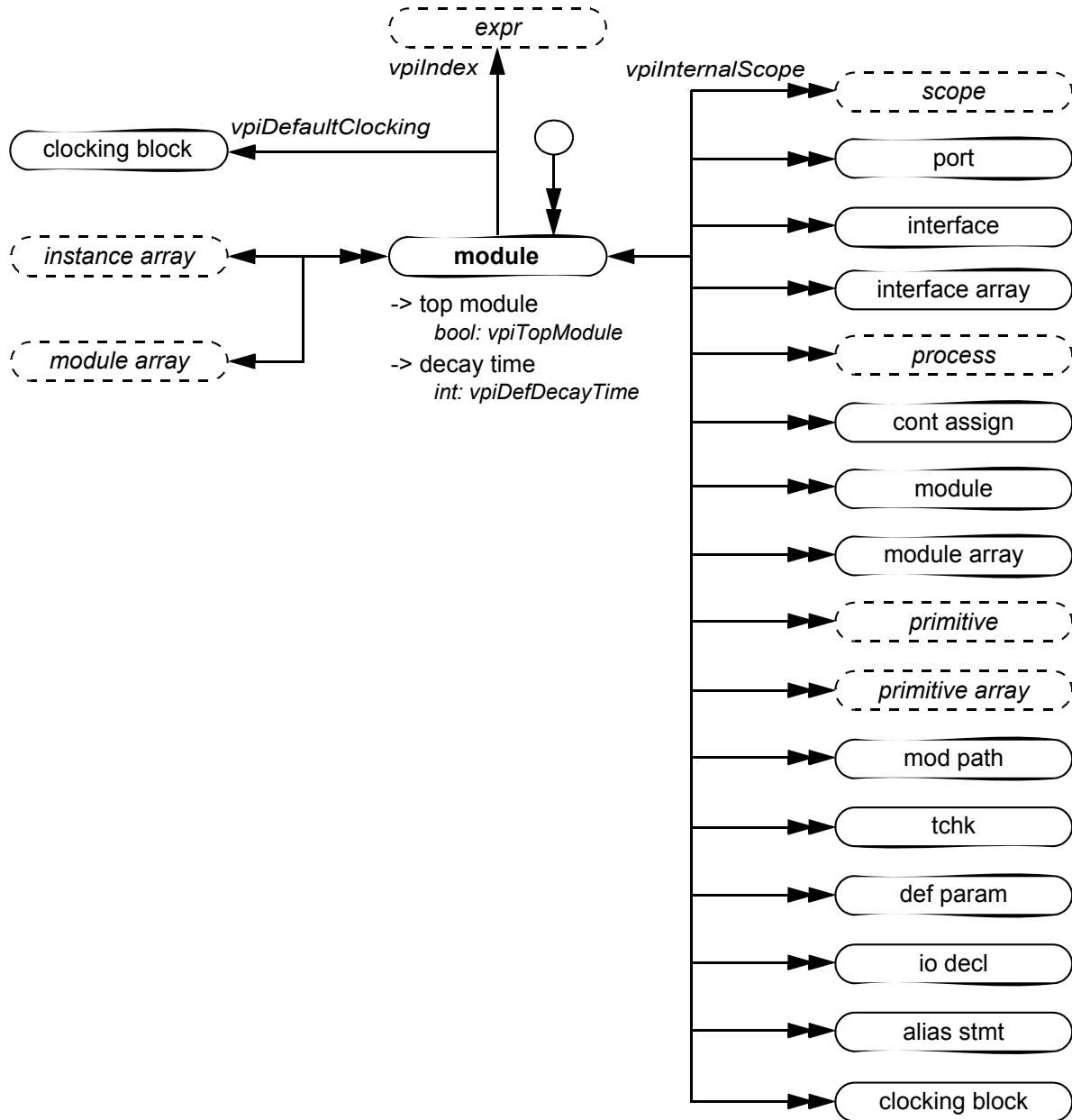
Details:

a)   Top-level modules shall be accessed using **vpi_iterate()** with a NULL reference object.

b)   If a module is an element within a module array, the **vpiIndex** transition is used to access the index within the array. If a module is not part of a module array, this transition shall return NULL.

27.3 Interface
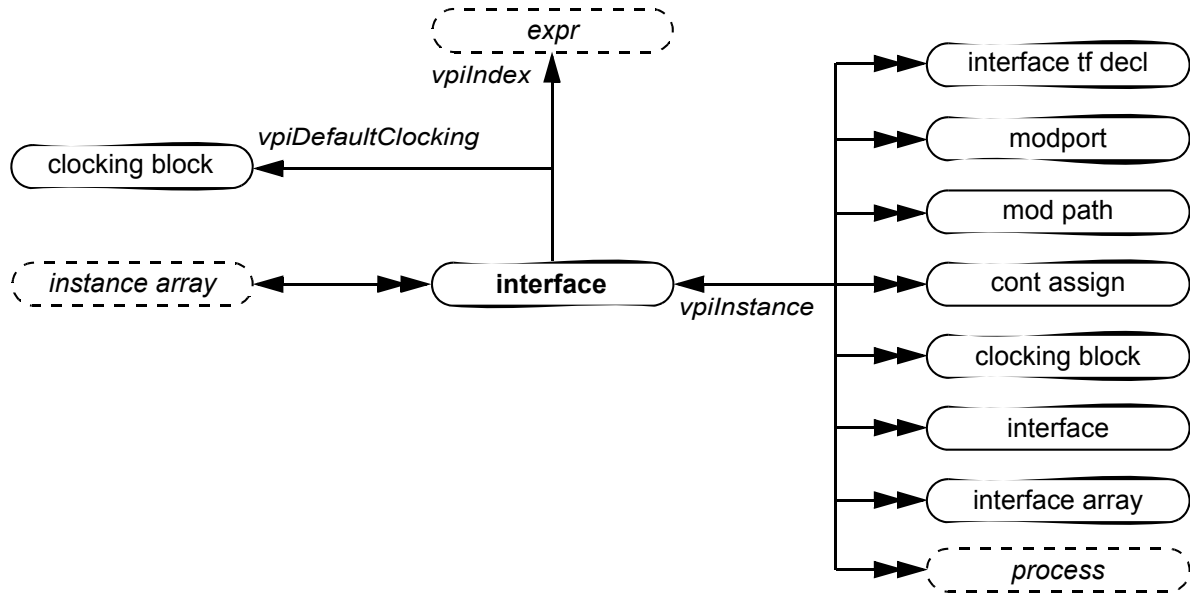
Details:

If an interface is an element within an instance array, the **vpiIndex** transition is used to access the index within the array. If an interface is not part of an instance array, this transition shall return NULL.

27.4 Modport
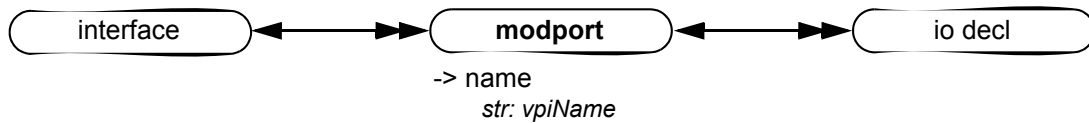
27.5 Interface task and function declaration
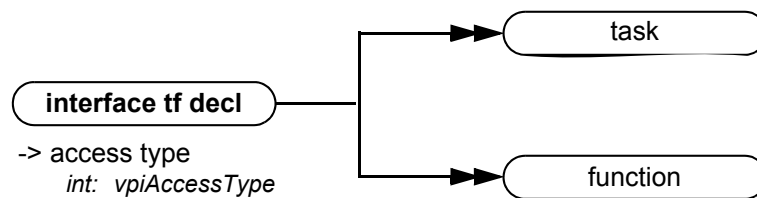
Details:

    a)   **vpi_iterate()** can return more than one task or function declaration for modport tasks and functions with an access type of **vpiForkJoin** because the task or function can be imported from multiple module instances.

    b)   Possible return values for the **vpiAccessType** property for an interface tf decl are **vpiForkJoin** and **vpiExtern**.

27.6 Program
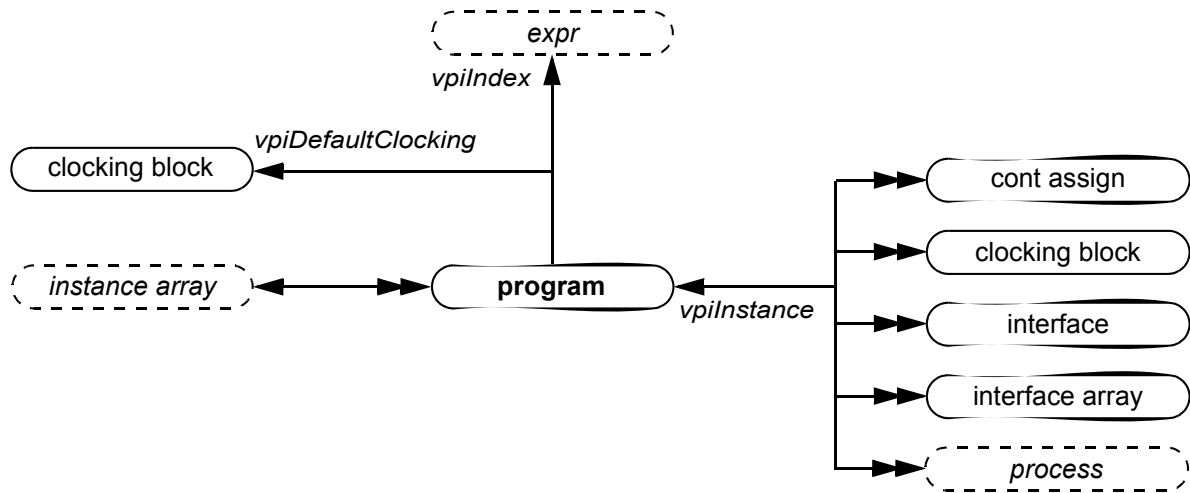
Details:

If a program is an element within an instance array, the **vpiIndex** transition is used to access the index within the array. If a program is not part of an instance array, this transition shall return NULL.

27.7 Instance
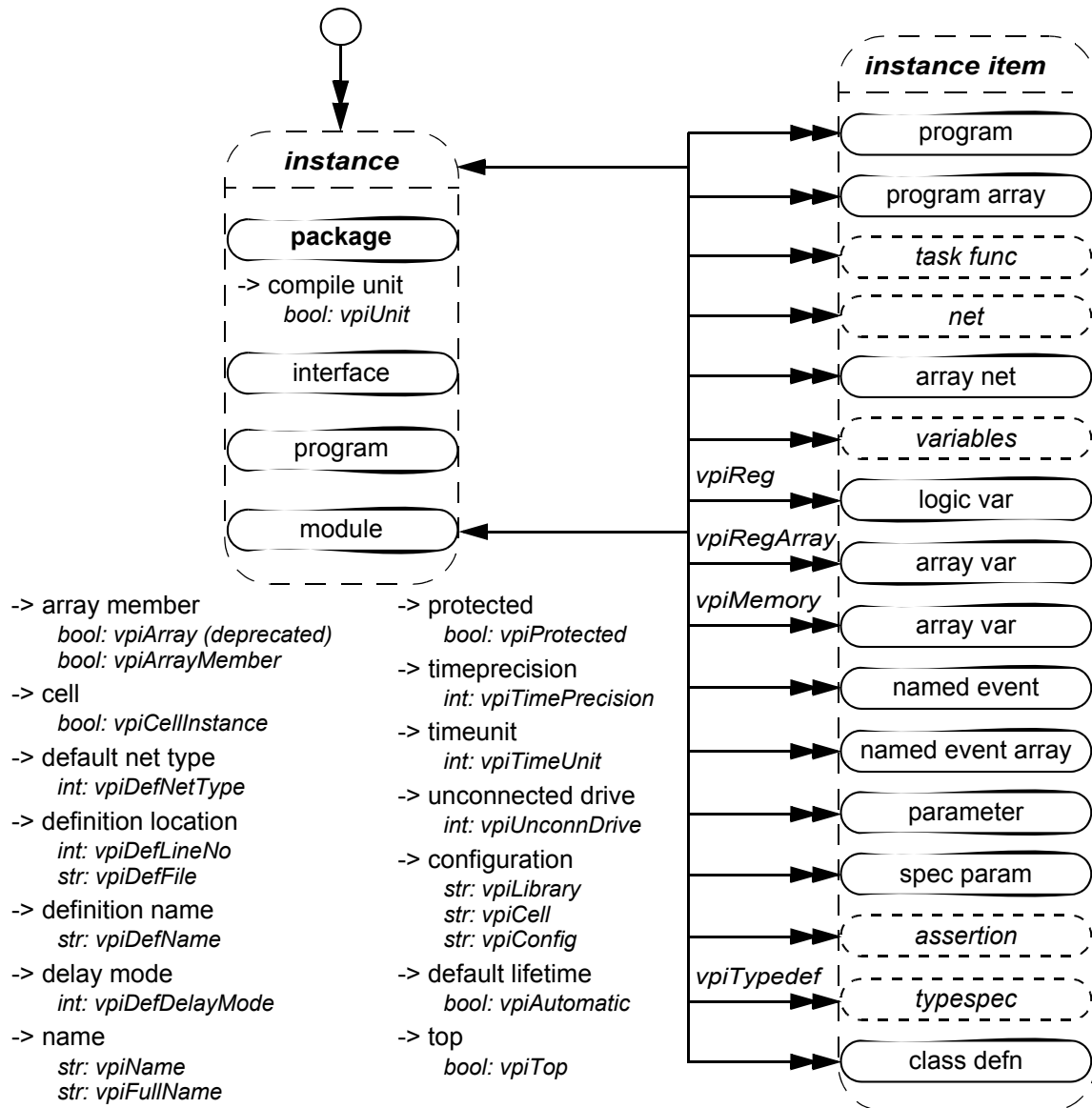
Details:

a) The **vpiTypedef** iteration shall return the user-defined typespecs that have typedefs explicitly declared in the instance.

b) **vpiModule** shall return a module if the object is inside a module instance; otherwise, it shall return NULL.

c) **vpiInstance** shall always return the immediate instance (package, module, program, or interface) in which the object is instantiated.

d) **vpiMemory** shall return array variable objects rather than **vpiMemory** objects. IEEE Std 1364 has made a similar update to the Verilog VPI (refer to detail 1 in 26.6.9 of IEEE Std 1364).

e) **vpiFullName** for objects that exist within a compilation unit shall begin with '$unit::' and, therefore, may be ambiguous. **vpiFullName** for a package shall be the name of the package and should end with "::"; this syntax disambiguates between a module and a package of the same name.

vpiFullName for objects that exist in a package shall begin with the name of the package followed by "`::`". The separator `::` shall appear between the package name and the immediately following name component. The `.` separator shall be used in all cases except package and class defn.

f) The following items shall not be accessible via `vpi_handle_by_name()`:

— Imported items

— Objects that exist within a compilation unit

g) Passing a `NULL` handle to **vpi_get()** with properties **vpiTimePrecision** or **vpiTimeUnit** shall return the smallest time precision of all modules in the instantiated design.

h) The properties **vpiDefLineNo** and **vpiDefFile** can be affected by the `'line` compiler directive. See 19.7 of IEEE Std 1364 for more details on the `'line` directive.

27.8 Instance arrays (supersedes 26.6.2 of IEEE Std 1364)


```
-> access by index
      vpi_handle_by_index()
      vpi_handle_by_multi_index()
-> name
      str: vpiName
      str: vpiFullName
-> size
      int: vpiSize
```
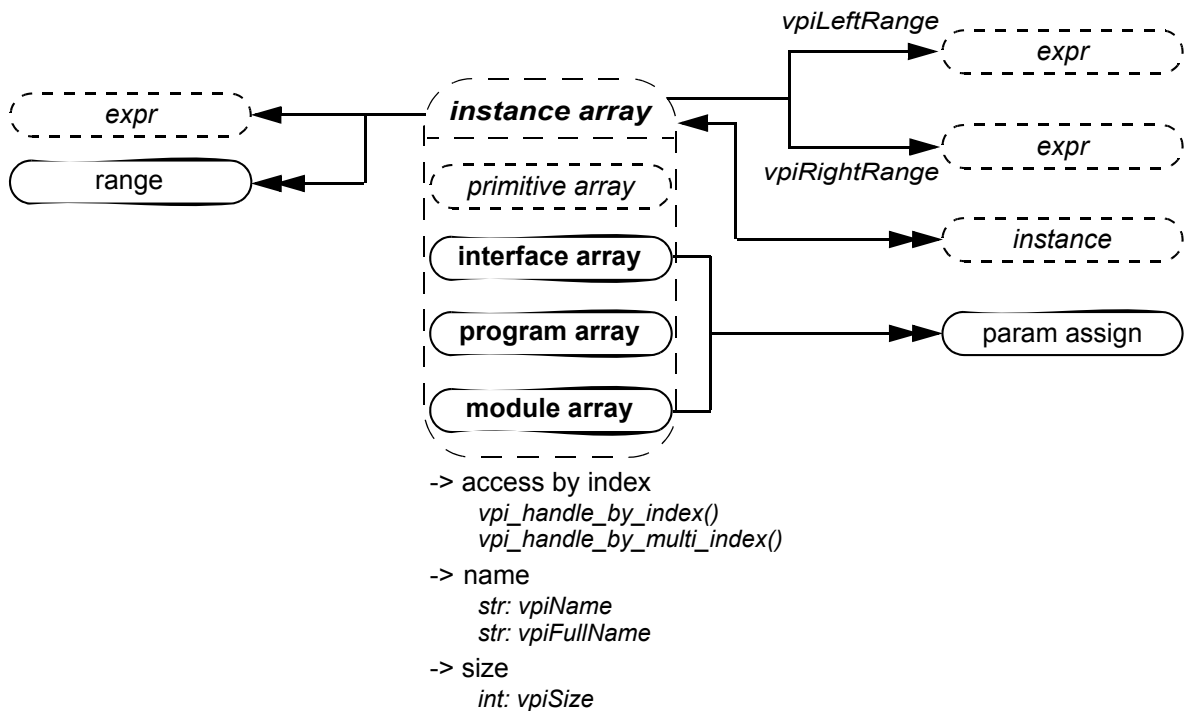
Details:

a) Traversing from the instance array to expr shall return a simple expression object of type **vpiOperation** with a **vpiOpType** of **vpiListOp**. This expression can be used to access the actual list of connections to the instance array in the Verilog source code.

b) To obtain all the dimensions of a multidimensional array, the range iterator must be used. Using the **vpiLeftRange**/**vpiRightRange** properties only returns the last dimension of a multidimensional array.

27.9 Scope (supersedes 26.6.3 of IEEE Std 1364)
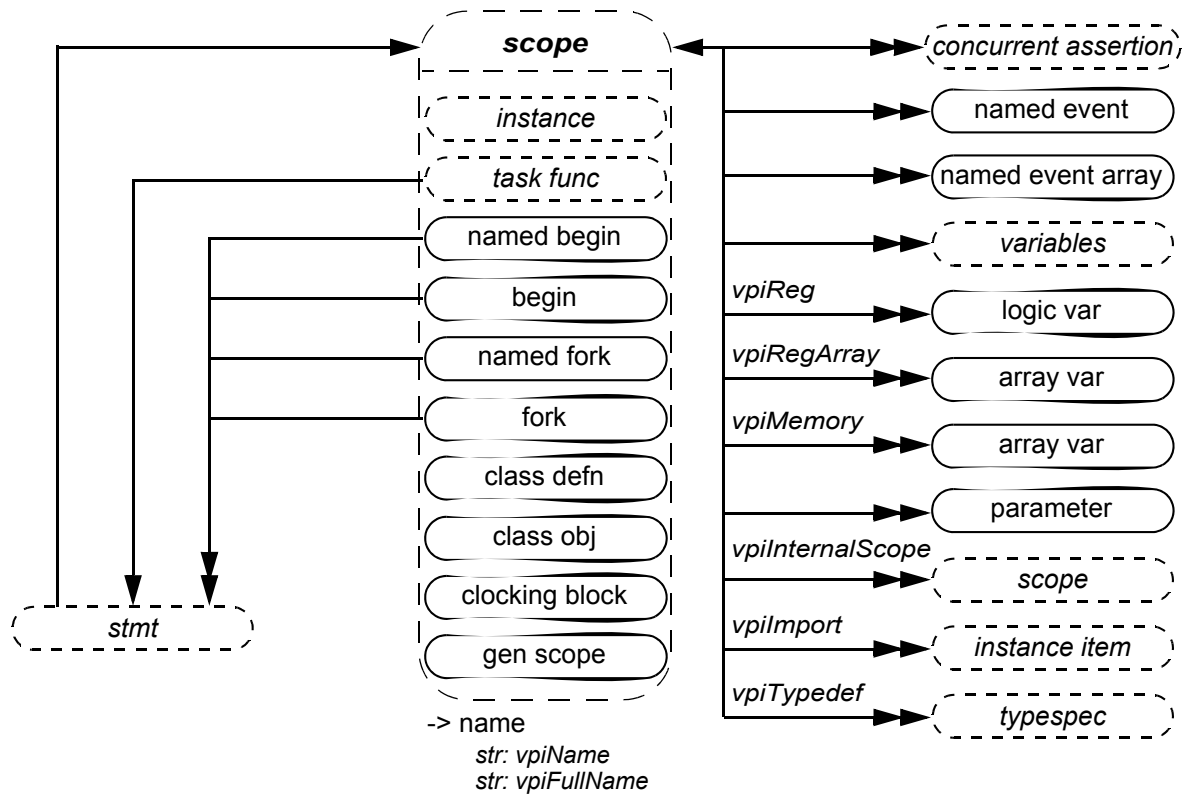
Details:

a) Unnamed scopes shall have valid names, although tool dependent.

b) The **vpiImport** iterator shall return all objects imported into the current scope via import statements. Only objects actually referenced through the import shall be returned, rather than items potentially made visible as a result of the import. Refer to 19.2.2 for more details.

c) A task func can have zero or more statements (see 12.2 and 12.3). If the number of statements is greater than 1, the **vpiStmt** relation shall return an unnamed **begin** that contains the statements of the task or function. If the number of statements is zero, the **vpiStmt** relation shall return NULL.

27.10 IO declaration (supersedes 26.6.4 of IEEE Std 1364)
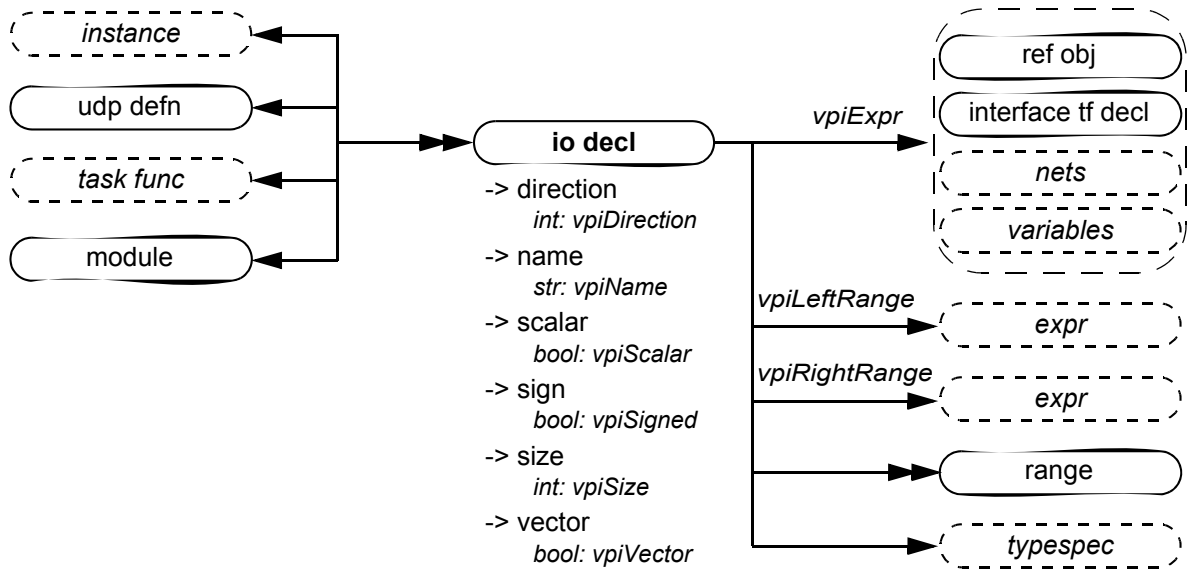
Details:

   a)   **vpiDirection** returns **vpiRef** for pass by `ref` ports or arguments.

   b)   A ref obj type handle may be returned for the **vpiExpr** of an io decl if it is passed by reference or if the io decl is an interface or a modport.

   c)   If the **vpiExpr** of an io decl is a ref obj and if the **vpiActual** of the ref obj is an interface or modport declaration, then the **vpiDirection** of the io decl shall be undefined.

27.11 Ports (supersedes 26.6.5 of IEEE Std 1364)

Details:

a) **vpiPortType** shall be one of the following three types: **vpiPort**, **vpiInterfacePort**, and **vpiMod-portPort.** Port type depends on the formal, not on the actual.

b) **vpi_get_delays(), vpi_put_delays()** delays shall not be applicable for **vpiInterfacePort.**

c) **vpiHighConn** shall indicate the hierarchically higher (closer to the top module) port connection.

d) **vpiLowConn** shall indicate the lower (further from the top module) port connection.

e) **vpiLowConn** of a **vpiInterfacePort** shall always be **vpiRefObj**.

f) Properties **vpiScalar** and **vpiVector** shall indicate if the port is 1 bit or more than 1 bit. They shall not indicate anything about what is connected to the port.

g) Properties **vpiIndex** and **vpiName** shall not apply for port bits.

h) If a port is explicitly named, then the explicit name shall be returned. If not, and a name exists, then that name shall be returned. Otherwise, NULL shall be returned.

i) **vpiPortIndex** can be used to determine the port order. The first port has a port index of zero.

j) **vpiLowConn** shall return NULL if the module or interface or program port is a null port (e.g., "module M();"). **vpiHighConn** shall return NULL if the instance of the module, interface, or program does not have a connection to the port.

k) **vpiSize** for a null port shall return 0.

27.12 Reference objects
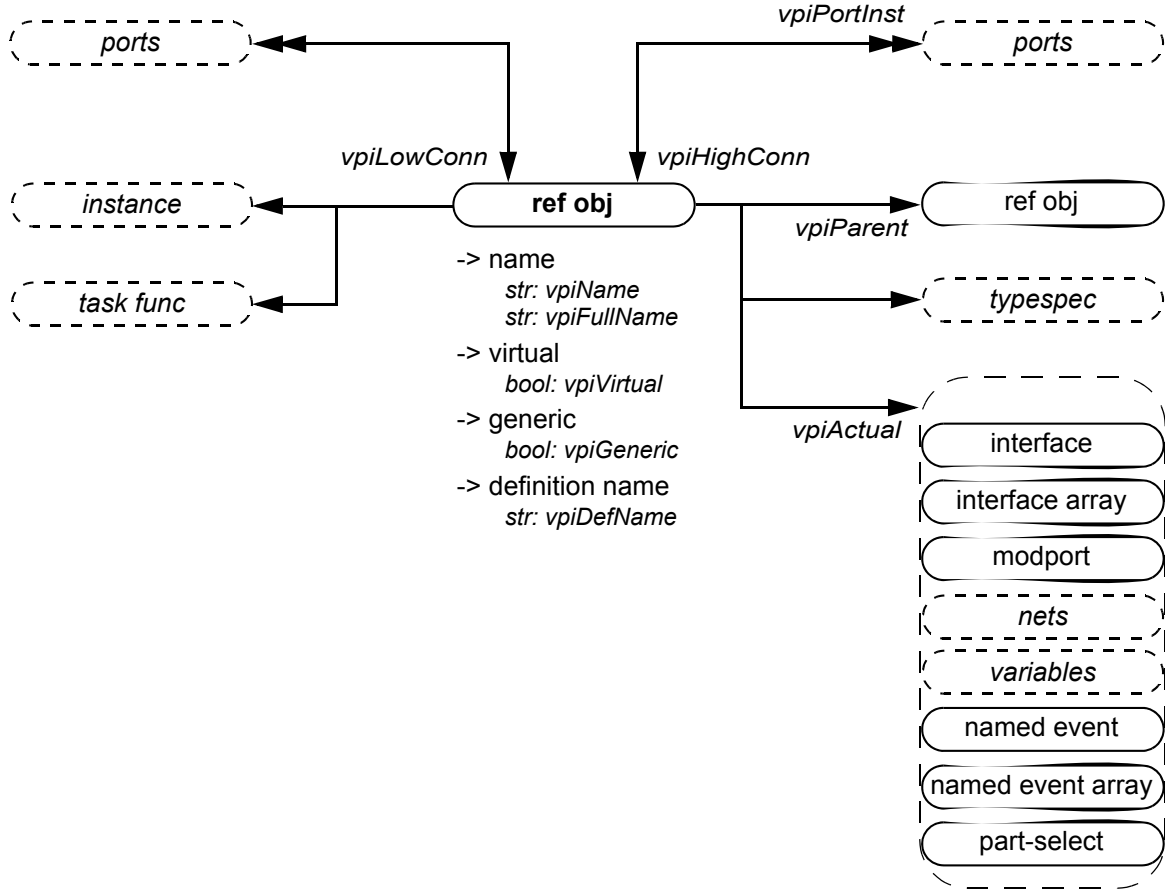
Details:

a) A ref obj represents a declared object or subelement of that object that is a reference to an actual instantiated object. A ref obj exists for ports with **ref** direction, for an interface port, for a modport port, or for formal task function ref arguments. The specific cases for a ref obj are as follows:

— A variable, named event, or named event array that is the lowconn of a ref port

— Any subelement expression of the above

— A local declaration of an interface or modport passed through a port or any net, variable, named event, or named event array of those

— A virtual interface declaration in a class definition

— A ref formal argument of a task or function or of a subelement expression of it

A ref obj may be obtained when walking port connections (lowConn, highConn), when traversing an expression that is a use of such ref obj, when accessing the virtual interface of a class, or when accessing the io decl of an instance, task, or function.

b) The name of ref obj can be different at every instance level it is being declared. The **vpiActual** relationship always returns the actual instantiated object if the ref obj is bound to an actual object at the time of the query.

c) The **vpiParent** relationship allows the traversal of a ref obj that is a subelement of a ref obj. In the example below, `r[0]` is a ref obj whose parent is the ref obj `r`. The **vpiActual** for the ref obj `r[0]` would return the var bit `a[0]`, and the **vpiActual** of the ref obj `r` would return the variable `a`.

```
        module top;
            logic [2:0] a;
            u1 m (a);
        endmodule
        module n (ref [2:0] r);
            initial
                r[0] = 1'b0;
        endmodule
```

d) The **vpiVirtual** property shall return TRUE if the ref obj is a reference to a virtual interface and FALSE if the ref obj is a reference to an interface that is not a virtual interface. The **vpiVirtual** property shall return **vpiUndefined** for all other kinds of ref obj.

e) The **vpiGeneric** property shall return TRUE if the ref obj is a reference to a generic interface and FALSE if the ref obj is a reference to an interface that is not a generic interface. The **vpiGeneric** property shall return **vpiUndefined** for all other kinds of ref obj.

f) The **vpiDefName** property when applied to a ref obj that is an actual of an interface or modport shall return the interface definition name or modport name.

g) The **vpiTypeSpec** property returns NULL for a ref obj of which **vpiActual** is a not a net, variable, or part-select.

27.12.1 Examples

Example 1: Passing an interface or modport through a port:

```
interface simple ();
    logic req, gnt;
    modport slave (input req, output gnt);
    modport master (input gnt, output req);
endinterface

module top();

    interface simple i;

    child1 i1(i);
    child2 i2(i.master);
endmodule

/***********************************
for the port of i1,
    the vpiHighConn relationship returns a handle of type vpiRefObj. The
    vpiActual relationship applied to the ref obj returns a handle of type
    vpiInterface.

for the port of i2,
    the vpiHighConn relationship returns a handle of type vpiRefObj. The
    vpiActual relationship applied to the ref obj returns a handle of type
    vpiModport.
***************************************/

module child1(interface simple s);
    c1 c_1(s);
    c1 c_2(s.master);
endmodule

/**************************
```

```
        for the port of module child1,
           the vpiLowConn relationship returns a handle of type vpiRefObj. The
           vpiActual relationship applied to the ref obj returns a handle of type
           vpiInterface.
        for that refObj,
           the vpiPort relationship returns the port of child1.
           the vpiPortInst iteration returns handles to s, s.master.
           the vpiActual relationship returns a handle to i.
        for the port of instance c_1:
           vpiHighConn returns a handle of type vpiRefObj. The vpiActual relationship
           applied to the ref obj handle returns a handle of type vpiInterface.
        for the port of instance c_2:
           vpiHighConn returns a handle of type vpiRefObj. The vpiActual relationship
           applied to the ref obj handle returns a handle of type vpiModport.
        *****************************************/
```

Example 2: Virtual interface declaration in a class definition:

```
        interface SBus; // A Simple bus interface
           logic req, grant;
           logic [7:0] addr, data;
        endinterface

        class SBusTransactor; // SBus transactor class
           virtual SBus bus; // virtual interface of type SBus
           function new( virtual SBus s );
              bus = s; // initialize the virtual interface
           endfunction
           task request(); // request the bus
              bus.req <= 1'b1;
           endtask
           task wait_for_bus(); // wait for the bus to be granted
              @(posedge bus.grant);
           endtask
        endclass

        module devA( Sbus s ); ... endmodule // devices that use SBus

        module devB( Sbus s ); ... endmodule

        module top;
           SBus s[1:4] (); // instantiate 4 interfaces
           devA a1( s[1] ); // instantiate 4 devices
           devB b1( s[2] );
           devA a2( s[3] );
           devB b2( s[4] );
           initial begin
              SbusTransactor t[1:4]; // create 4 bus-transactors and bind
              t[1] = new( s[1] );
              t[2] = new( s[2] );
              t[3] = new( s[3] );
              t[4] = new( s[4] );
           end
        endmodule
```

A ref obj is returned for the left-hand expression of the statement "bus = s" in the constructor of the class definition SBustransactor. The **vpiName** of that ref obj is "bus", and its **vpiDefName** is the name of the interface "SBus". The **vpiActual** relationship returns the interface instance associated with that particular

call to new after the assignment has executed. For example, if it was "`new ( s[1] )`", **vpiActual** would return the interface `s[1]`. If **vpiActual** is queried before the assignment is executed, the method may return `NULL` if the virtual "`bus`" interface is uninitialized. The right-hand expression also returns a ref obj which **vpiActual** is the interface instance passed to the call to new.

27.13 Nets (supersedes 26.6.6 of IEEE Std 1364)

-> access by index
  vpi_handle_by_index()
  vpi_handle_by_multi_index()
-> array member
  bool: vpiArray (deprecated)
  bool: vpiArrayMember
-> delay
  vpi_get_delays()
-> expanded
  bool: vpiExpanded
-> implicitly declared
  bool: vpiImplicitDecl
-> name
  str: vpiName
  str: vpiFullName

-> net decl assign
  bool: vpiNetDeclAssign
-> net type
  int: vpiNetType
  int: vpiResolvedNetType
-> scalar
  bool: vpiScalar
-> scalared declaration
  bool: vpiExplicitScalared
-> sign
  bool: vpiSigned
-> size
  int: vpiSize

-> strength
  int: vpiStrength0
  int: vpiStrength1
  int: vpiChargeStrength
-> value
  vpi_get_value()
  vpi_put_value()
-> vector
  bool: vpiVector
-> vectored declaration
  bool: vpiExplicitVectored
-> member
  bool: vpiStructUnionMember

Details:

a) Any net declared as an array with one or more unpacked ranges is an array net. The range iterator for an array net returns only the unpacked ranges for the array.

b) The boolean property **vpiArray** is deprecated in this standard. The **vpiArrayMember** property shall be TRUE for a net that is an element of an array net. It shall be FALSE otherwise.

c) For logic nets, net bits shall be available regardless of vector expansion.

d) Continuous assignments and primitive terminals shall be accessed regardless of hierarchical boundaries.

e) Continuous assignments and primitive terminals shall only be accessed from scalar nets or bit-selects.

f) For **vpiPorts**, if the reference handle is a net bit, then port bits shall be returned. If it is an entire net or array net, then a handle to the entire port shall be returned.

g) For **vpiPortInst**, if the reference handle is a bit or scalar, then port bits or scalar ports shall be returned, unless the highconn for the port is a complex expression where the bit index cannot be determined. If this is the case, then the entire port shall be returned. If the reference handle is an entire net or array net, then the entire port shall be returned.

h) For **vpiPortInst**, it is possible for the reference handle to be part of the highconn expression, but not connected to any of the bits of the port. This may occur if there is a size mismatch. In this situation, the port shall not qualify as a member for that iteration.

i) For implicit nets, vpiLineNo shall return 0, and **vpiFile** shall return the file name where the implicit net is first referenced.

j) **vpi_handle(vpiIndex, net_bit_handle)** shall return the bit index for the net bit. **vpi_iterate (vpiIndex, net_bit_handle)** shall return the set of indices for a multidimensional net array bit-select, starting with the index for the net bit and working outward.

k) Only active forces and assign statements shall be returned for **vpiLoad**.

l) Only active forces shall be returned for **vpiDriver**.

m) **vpiDriver** shall also return ports that are driven by objects other than nets and net bits.

n) **vpiLocalLoad** and **vpiLocalDriver** return only the loads or drivers that are local, i.e., contained by the module instance that contains the net, including any ports connected to the net (output and inout ports are loads, input and inout ports are drivers).

o) For **vpiLoad**, **vpiLocalLoad**, **vpiDriver**, and **vpiLocalDriver** iterators, if the object is **vpiNet** for an enum net, an integer net, or a time net or for a logic net or struct net for which **vpiVector** is TRUE, then all loads or drivers are returned exactly once as the loading or driving object. In other words, if a part-select loads or drives only some bits, the load or driver returned is the part-select. If a driver is repeated, it is only returned once. To trace exact bit-by-bit connectivity, pass a **vpiNetBit** object to **vpi_iterate**.

p) An iteration on loads or drivers for a variable bit-select shall return the set of loads or drivers for whatever bit to which the bit-select is referring at the beginning of the iteration.

q) **vpiSimNet** shall return a unique net if an implementation collapses nets across hierarchy (refer to 12.3.10 of IEEE Std 1364 for the definition of simulated net and collapsed net).

r) The property **vpiExpanded** on an object of type **vpiNetBit** shall return the property's value for the parent.

s) The loads and drivers returned from **(vpiLoad, obj_handle)** and **vpi_iterate(vpiDriver, obj_handle)** may not be the same in different implementations, due to allowable net collapsing (see 12.3.10 of IEEE Std 1364). The loads and drivers returned from **vpi_iterate(vpiLocalLoad, obj_handle)** and **vpi_iterate(vpiLocalDriver, obj_handle)** shall be the same for all implementations.

t) The boolean property **vpiConstantSelect** returns TRUE if the expression that constitutes the index or indices evaluates to a constant and FALSE otherwise.

u) **vpiSize** for an array net shall return the number of nets in the array. For unpacked structures, the size returned indicates the number of members in the structure. For an enum net, integer net, logic net, time net, or packed struct net, **vpiSize** shall return the size of the net in bits. For a net bit, **vpiSize** shall return 1.

v) **vpi_iterate(vpiIndex, net_handle)** shall return the set of indices for a net within an array net, starting with the index for the net and working outward. If the net is not part of an array (the **vpiArrayMember** property is FALSE), a NULL shall be returned.

w) **vpi_iterate(vpiRange, array_net_handle)** shall return the set of array range declarations beginning with the leftmost range of the array declaration and iterate to the rightmost range of the array declaration.

x) **vpiArrayNet** is #defined the same as **vpiNetArray** for backward compatibility. A call to **vpi_get_str(vpiType, <array_net_handle>)** may return either "vpiArrayNet" or "vpiNetArray".

y) A logic net without a packed dimension defined is a scalar; and for that object the property **vpiScalar** shall return TRUE, and the property **vpiVector** shall return FALSE. A logic net with one or more packed dimensions defined is a vector, and the property **vpiVector** shall return TRUE (**vpiScalar** shall return FALSE). A packed struct net is a vector, and the property **vpiVector** shall return TRUE (**vpiScalar** shall return FALSE). A net bit is a scalar, and the property **vpiScalar** shall return TRUE (**vpiVector** shall return FALSE). The properties **vpiScalar** and **vpiVector** when queried on a handle to an enum net shall return the value of the respective property for an object for which the typespec is the same as the base typespec of the typespec of the enum net. For an integer net or a time net, the property **vpiVector** shall return TRUE (**vpiScalar** shall return FALSE). For an array net, the **vpiScalar** and **vpiVector** properties shall return the values of the respective properties for an array element. The **vpiScalar** and **vpiVector** properties shall return FALSE for all other net objects.

z) **vpiLogicNet** is #defined the same as **vpiNet** for backward compatibility. A call to **vpi_get_str (vpiType, <logic_net_handle>)** may return either "vpiLogicNet" or "vpiNet".

aa) Neither an array net nor an unpacked struct net has a value property.

27.14 Variables (supersedes 26.6.7 and 26.6.8 of IEEE Std 1364)

A value of **vpiValidTrue** for the property **vpiValid** shall indicate that the application may continue to access the properties, relationships, and value of the variable indicated by the reference handle. A value of **vpiValidFalse** shall indicate that the variable indicated by the reference handle can no longer be accessed. A variable may cease to exist, for example, if the scope terminates in which an automatic variable is declared (see 6.6) or because the object is reclaimed by automatic memory management (see 7.26).

If an implementation is unable to determine whether the associated variable still exists, it may cause **vpiValid** to return a value of **vpiValidUnknown**. In this case, the application may attempt to reacquire a handle to the variable by starting from a static handle or from a handle for which **vpiValid** returns **vpiValidTrue**.

NOTE—An attempt to reacquire a handle to a variable for which **vpiValid** returns **vpiValidFalse** will always fail. An attempt to reacquire a handle to a variable for which **vpiValid** returns **vpiValidTrue** is unnecessary, but will always succeed.

It shall be an error for an application to attempt to obtain a property, relationship, or value of an invalid variable.

Details:

a) Any variable declared as an array with one or more unpacked ranges is an array var. The range iterator for an array var returns only the unpacked ranges for the array.

b) The boolean property **vpiArray** is deprecated in this standard. The boolean property **vpiArrayMember** shall be TRUE if the referenced variable is a member of an array variable. It shall be FALSE otherwise.

c) To obtain the members of a union and structure, see the relations in 27.18

d) The range relation is valid only when the variable is an array var or when the variable is a logic var or a bit var and the property **vpiVector** is TRUE. When applied to array vars, this relation returns only unpacked ranges. When applied to logic and bit variables, it returns only the packed ranges.

e) **vpi_handle(vpiIndex, var_select_handle)** shall return the index of a var select in a one-dimensional array. **vpi_iterate(vpiIndex, var_select_handle)** shall return the set of indices for a var select in a multidimensional array, starting with the index for the var select and working outward.

f) If a logic var or bit var has more than one packed dimension, **vpiLeftRange** and **vpiRightRange** shall return the bounds of the leftmost packed dimension. If an array var has more than one unpacked dimension, **vpiLeftRange** and **vpiRightRange** shall return the bounds of the leftmost unpacked dimension.

g) A var select is an element selected from an array var.

h) If the variable has an initialization expression, the expression can be obtained from **vpi_handle(vpiExpr, var_handle)**.

i) **vpiSize** for a variable array shall return the number of variables in the array. For nonarray variables, it shall return the size of the variable in bits. For unpacked structures and unions, the size returned indicates the number of fields in the structure or union.

j) **vpiSize** for a var select shall return the number of bits in the var select. This applies only for packed var select.

k) Variables of **vpiType**, **vpiArrayVar**, or **vpiClassVar** do not have a value property. Struct var and union var variables for which the **vpiVector** property is FALSE do not have a value property.

l) **vpiBit** iterator applies only for logic, bit, packed struct, and packed union variables.

m) **vpi_handle(vpiIndex, var_bit_handle)** shall return the bit index for the variable bit. **vpi_iterate (vpiIndex, var_bit_handle)** shall return the set of indices for a multidimensional variable bit-select, starting with the index for the bit and working outwards.

n) **cbSizeChange** shall be applicable only for dynamic and associative arrays. If both value and size change, the size change callback shall be invoked first. This callback fires after size change occurs and before any value changes for that variable. The value in the callback is the new size of the array.

o) The property **vpiRandType** returns the current randomization type for the variable, which can be one of **vpiRand**, **vpiRandC**, and **vpiNotRand**.

p) **vpiIsRandomized** is a property to determine whether a random variable is currently active for randomization.

q) When the **vpiStructUnionMember** property is TRUE, it indicates that the variable is a member of a parent struct or union variable. See also the relations in 27.18.

r) If a variable is an element of an array (the **vpiArrayMember** property is TRUE), the **vpiIndex** iterator shall return the indexing expressions that select that specific variable out of the array.

s) In the above diagram:

    logic var == reg
    var bit == reg bit
    array var == reg array

    vpiVarBit is #defined the same as **vpiRegBit** for backward compatibility. However, a **vpiVarBit** can be an element of a **vpiBitVar** (2-state) or a **vpiLogicVar** (4-state), whereas **vpiRegBit** could only be an element of a **vpiReg** (4-state).

    SystemVerilog treats `reg` and `logic` variables as equivalent in all respects. To allow for backward compatibility, **vpi_get_str(vpiType, <logic_var_handle>)** may return either "vpiLogicVar" or "vpiReg". Similarly, **vpi_get_str(vpiType, <var_bit_handle>)** may return either "vpiVarBit" or "vpiRegBit", while **vpi_get_str(vpiType, <array_var_handle>)** may return either "vpiArrayVar" or "vpiRegArray".

t) A bit var or logic var, without a packed dimension defined, is a scalar; and for those objects, the property **vpiScalar** shall return TRUE, and the property **vpiVector** shall return FALSE. A bit var or logic var, with one or more packed dimensions defined, is a vector, and the property **vpiVector** shall return TRUE (**vpiScalar** shall return FALSE). A packed struct var and a packed union var are vectors, and the property **vpiVector** shall return TRUE (**vpiScalar** shall return FALSE). A var bit is a scalar, and the property **vpiScalar** shall return TRUE (**vpiVector** shall return FALSE). The properties **vpiScalar** and **vpiVector** when queried on a handle to an enum var shall return the value of the respective property for an object for which the typespec is the same as the base typespec of the typespec of the enum var. For an integer var, time var, short int var, int var, long int var, and byte var, the property vpiVector shall return TRUE (**vpiScalar** shall return FALSE). For an array var, the **vpiScalar** and **vpiVector** properties shall return the values of the respective properties for an array element. The **vpiScalar** and **vpiVector** properties shall return FALSE for all other var objects.

u) **vpiArrayType** can be one of **vpiStaticArray**, **vpiDynamicArray**, **vpiAssocArray**, or **vpiQueue**.

v) **vpiRandType** can be one of **vpiRand**, **vpiRandC**, or **vpiNotRand**.

w) For more information on the **vpiAutomatic** lifetime property, refer to 26.6.20 of IEEE Std 1364.

x) **vpiVisibility** denotes the visibility (`local`, `protected`, or default) of a variable that is a class member. **vpiVisibility** shall return **vpiPublicVis** for a class member that is not `local` or `protected` or for a variable that is not a class member.

y) A nonstatic data member of a class var does not have a **vpiFullName** property. The static data member of a class, referenced either via a class var or a class defn, has the **vpiFullName** property. It shall return a full name string representing the hierarchical path of the static variable through "class defn". For example:

```
module top;

    class Packet ;
        static integer Id ;
        ....
```

```
    endclass

    Packet p;
    c = p.Id;
    ....
```

The **vpiFullName** for p.Id is "top.Packet::Id".

27.15 Variable select (supersedes 26.6.8 of IEEE Std 1364)
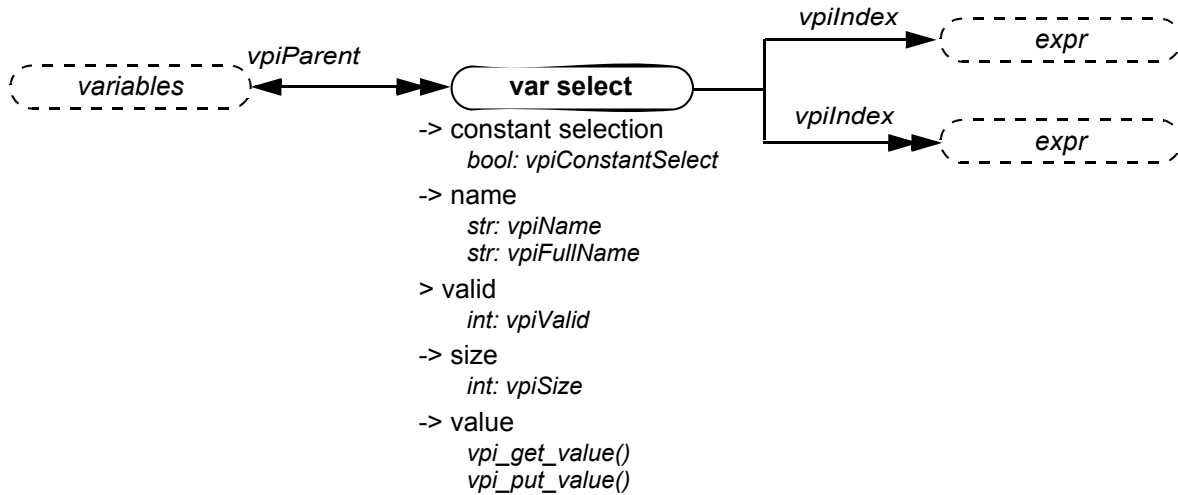
27.16 Variable drivers and loads (supersedes 26.6.23 of IEEE Std 1364)
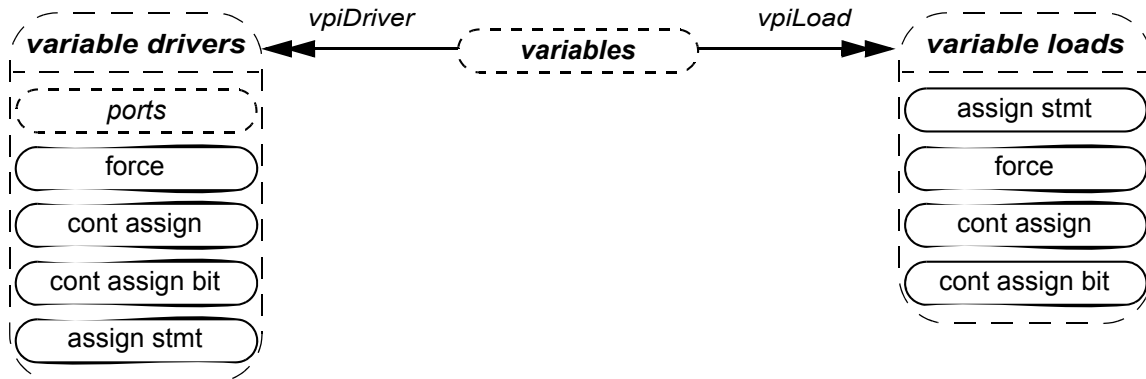
Details:

a) **vpiDrivers/Loads** for a structure, union, or class variable shall include the following:

    — Driver/Load for the whole variable

    — Driver/Load for any bit-select or part-select of that variable

    — Driver/Load of any member nested inside that variable

b) **vpiDrivers/Loads** for any variable array should include driver/load for entire array/vector or any portion of an array/vector to which a handle can be obtained.

27.17 Typespec
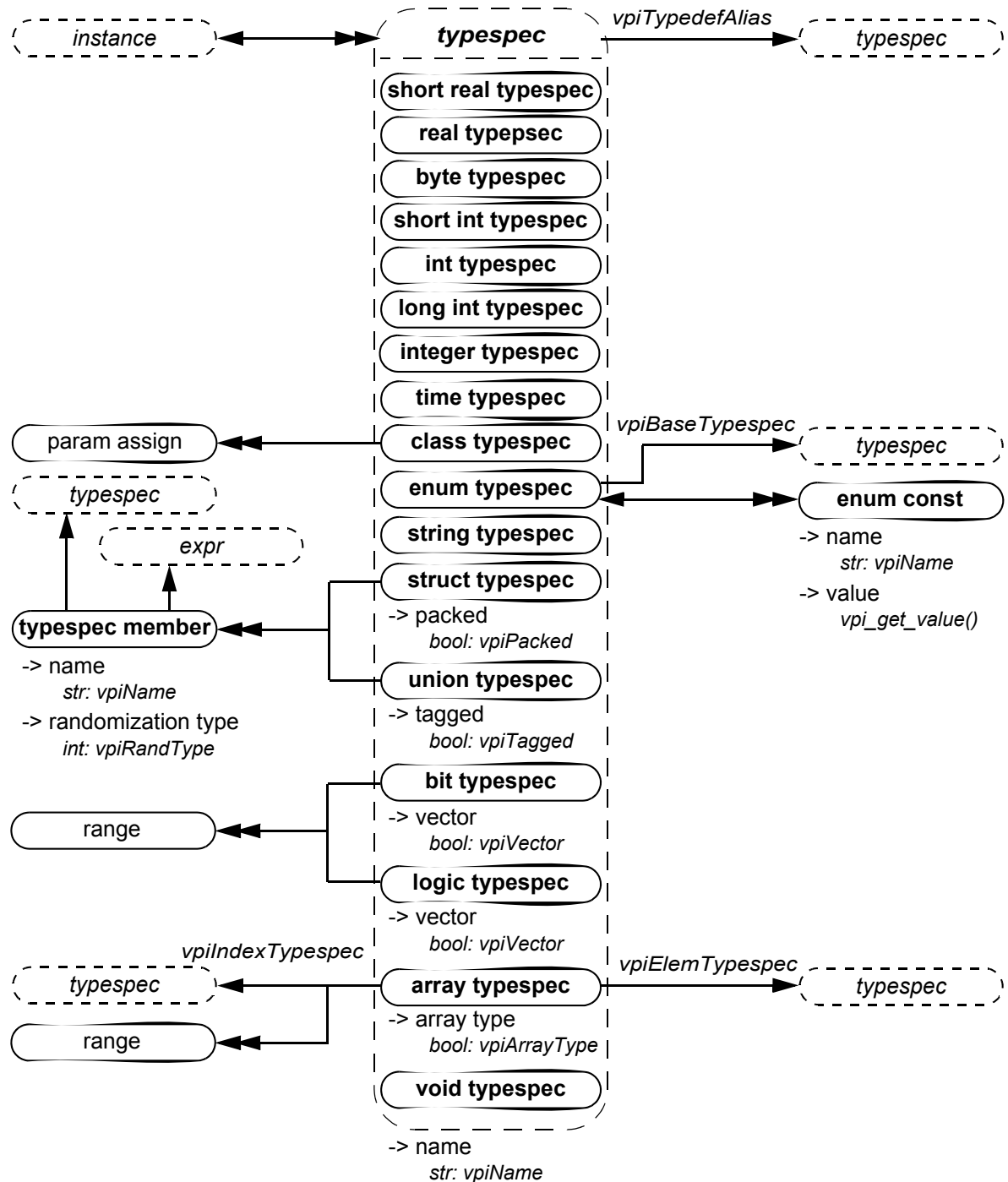
Details:

a) If a typespec denotes a type that has a user-defined typedef, the **vpiName** property shall return the name of that type; otherwise, the **vpiName** property shall return NULL. Consequently, the **vpiName** property returns NULL for any SystemVerilog built-in type. If the typespec denotes a type with a typedef that creates an alias of another typedef, then the **vpiTypedefAlias** of the typespec shall return a non-null handle, which represents the handle to the aliased typedef. For example:

```
typedef enum bit [0:2] {red, yellow, blue} primary_colors;
typedef primary_colors colors;
```

If "h1" is a handle to the typespec `colors`, its **vpiType** shall return **vpiEnumTypespec**, the **vpiName** property shall return "colors", and **vpiTypedefAlias** shall return a handle "h2" to the typespec "`primary_colors`" of **vpiType vpiEnumTypespec**. The **vpiName** property for "h2" shall return "`primary_colors`", and its **vpiTypedefAlias** shall return NULL.

b) **vpiIndexTypespec** relation is present only on associative arrays and returns the type that is used as the key into the associative array.

c) If the value of the property **vpiType** of a typespec is **vpiStructTypespec** or **vpiUnionTypespec**, then it is possible to iterate over **vpiTypespecMember** to obtain the structure of the user-defined type. For each typespec member, the typespec relation indicates the type of the member.

d) The property **vpiName** of a typespec member returns the name of the corresponding member, rather than the name (if any) of the associated typespec.

e) The name of a **typedef** may be the empty string if the typespec denotes typedef field defined in line rather than via a typedef declaration. For example:

```
typedef struct {
   struct
      int a;
   } B
} C;
```

The typespec representing the typedef `C` is a struct typespec; it has a single typespec member named `B`. The typespec relation for `B` returns another struct typespec that has no name and has a single typespec member named "a". The typespec relation for "a" returns an int typespec.

f) If a type is defined as an alias of another type, it inherits the **vpiType** of this other type. For example:

```
typedef time my_time;
my_time t;
```

The **vpiTypespec** of the variable named "t" shall return a handle h1 to the typespec "my_time" whose **vpiType** shall be a **vpiTimeTypespec**. The **vpiTypedefAlias** applied to handle h1 shall return a typespec handle h2 to the predefined type "**time**".

g) The expr associated with a typespec member shall represent the explicit default member value, if any, of the corresponding member of an unpacked structure data type (see 4.11).

27.18 Structures and unions
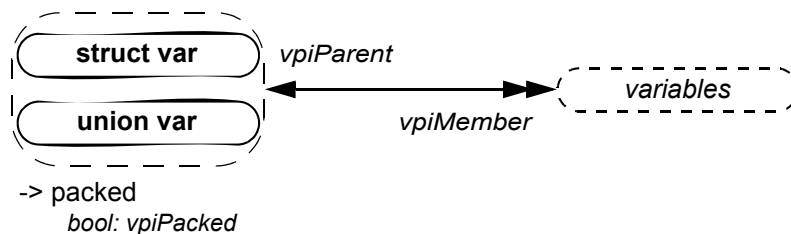
Details:

vpi_get_value()/vpi_put_value() cannot be used to access values of entire unpacked structures and unpacked unions.

27.19 Named events (supersedes 26.6.11 of IEEE Std 1364)
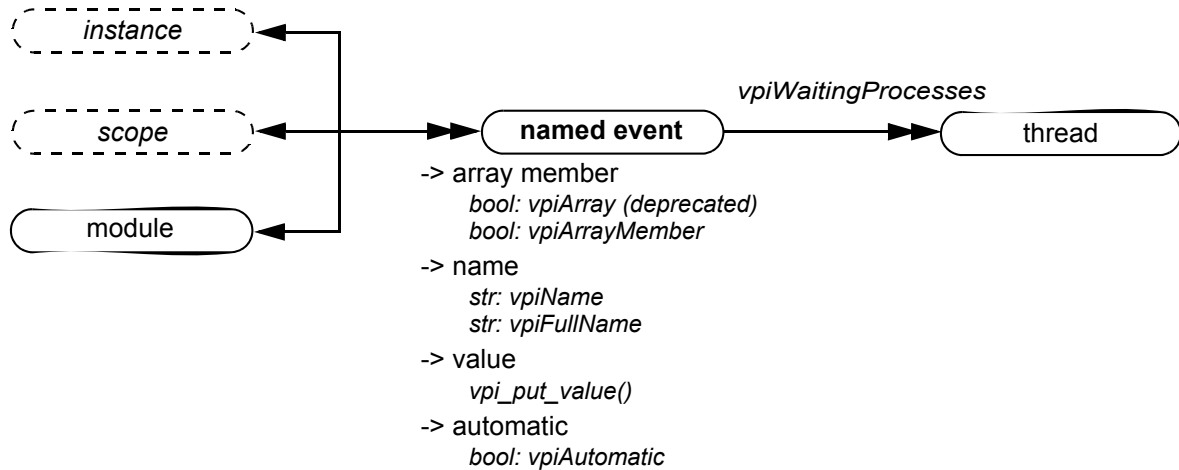
Details:

The **vpiWaitingProcesses** iterator returns all waiting processes, static or dynamic, identified by their thread, for that named event.
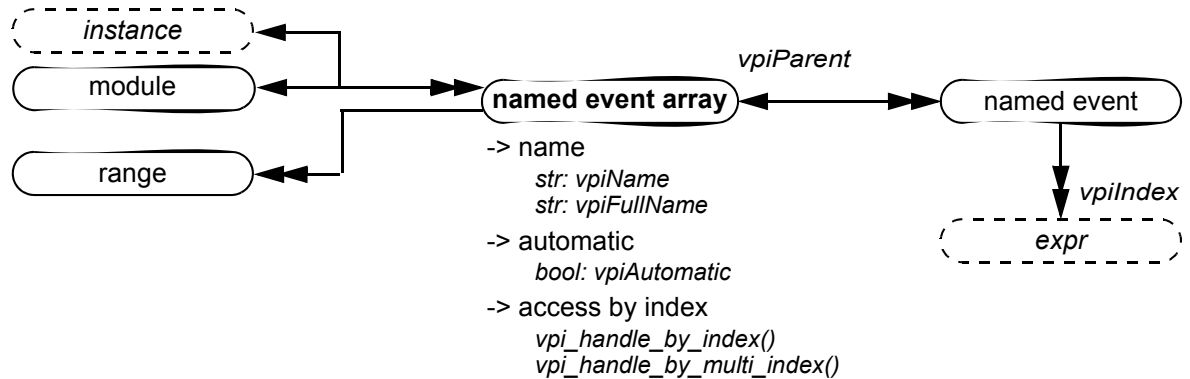
Details:

vpi_iterate(vpiIndex, named_event_handle) shall return the set of indices for a named event within an array, starting with the index for the named event and working outward. If the named event is not part of an array, a NULL shall be returned.

27.20 Parameter (supersedes 26.6.12 of IEEE Std 1364)
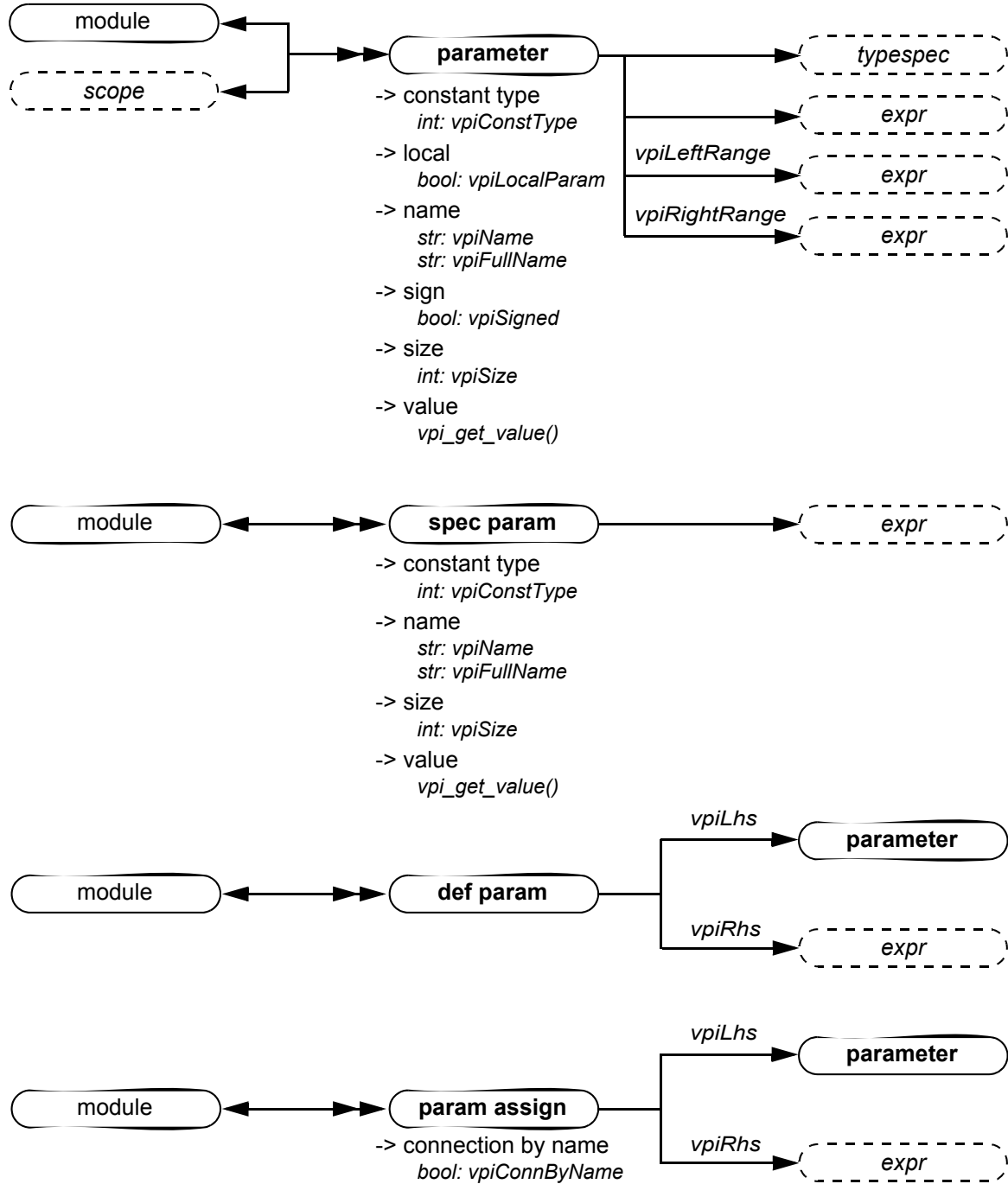
Details:

a) Obtaining the value from the object **parameter** shall return the final value of the parameter after all module instantiation overrides and defparams have been resolved.

b) **vpiLhs** from a param assign object shall return a handle to the overridden parameter.

c) If a parameter does not have an explicitly defined range or is a type parameter, **vpiLeftRange** and **vpiRightRange** shall return a NULL handle.

27.21 Class definition
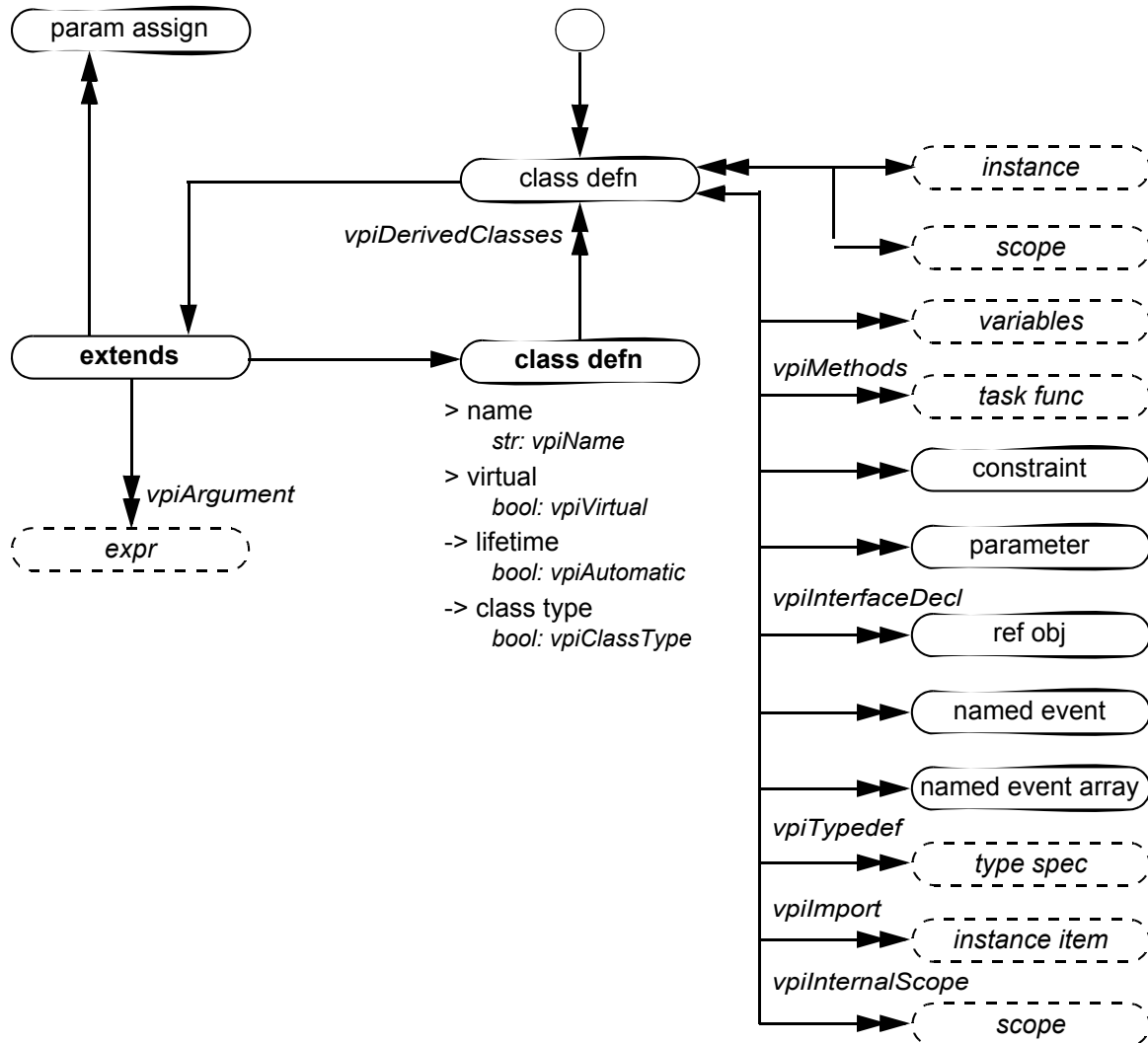
Details:

a) **vpi_get_value()** and **vpi_put_value()** are not allowed for nonstatic variable handles obtained from class defn handles.

b) The iterator to constraints returns only normal constraints and not inline constraints.

c) To get constraints inherited from base classes, it is necessary to traverse the extend relation to obtain the base class.

d) The **vpiDerivedClasses** iterator returns all the classes derived from the given class.

e) The relation to **vpiExtends** exists whenever one class is derived from another class (refer to 7.12). The relation from extends to class defn provides the base class. The iterators from extends to param assign and arguments provide the parameters and arguments used in constructor chaining (refer to 7.16 and 7.23).

f) The **vpiInterfaceDecl** iteration returns the virtual interface declarations in the class definition.

27.22 Class variables and class objects
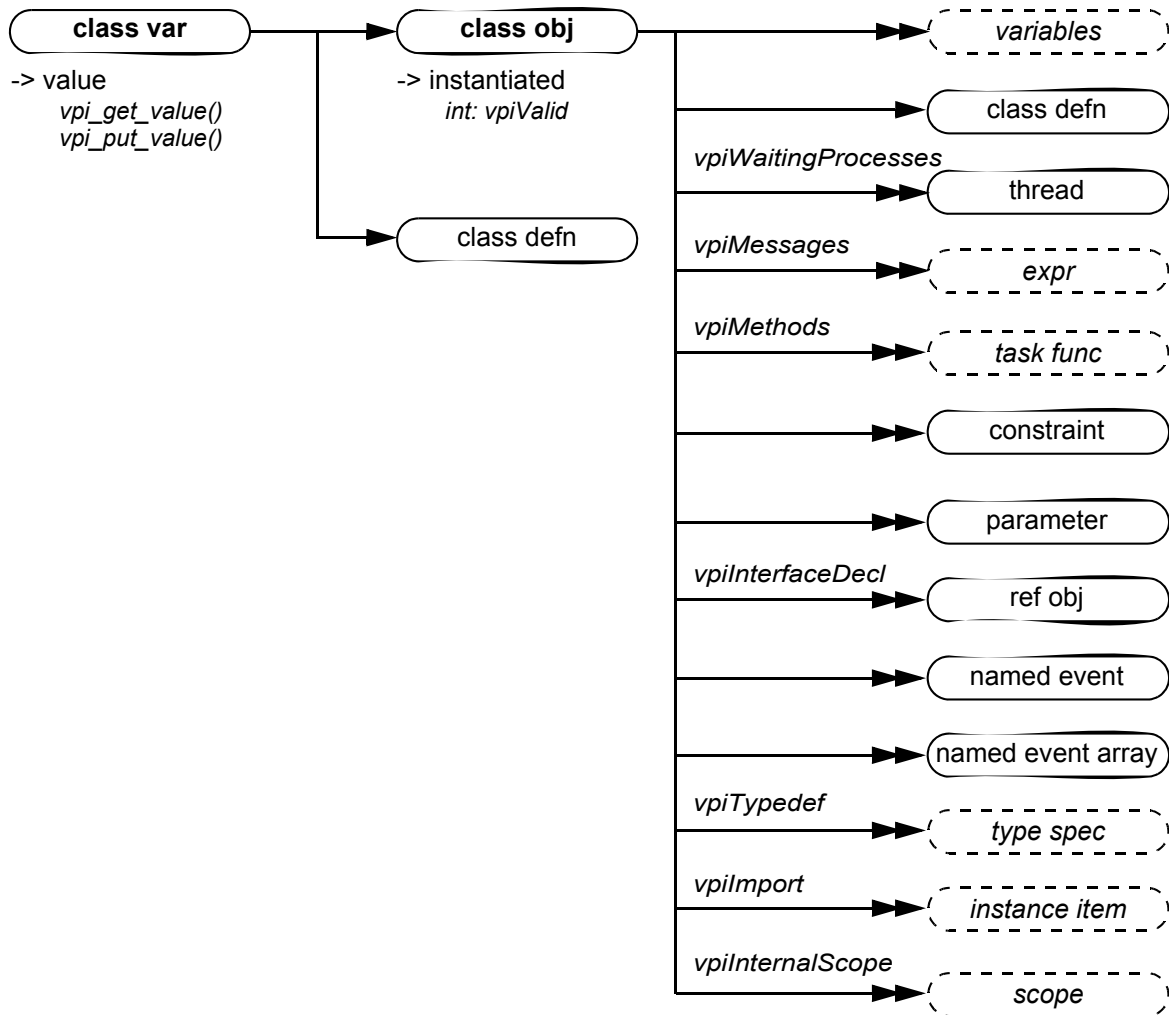
Details:

a) The **vpiWaitingProcesses** iterator on a mailbox or semaphore shall return the threads waiting on the class object or object resource. A waiting process is a static or dynamic process represented by its suspended thread. A process may be waiting to retrieve a message from a mailbox or waiting for a semaphore resource key.

b) **vpiMessages** iteration shall return all the messages in a mailbox.

c) For a class var, **vpiClassDefn** returns the class defn with which the class var was declared in the SystemVerilog source text. If the class var has the value of NULL, the **vpiClassObj** relationship applied to the class var shall return a null handle. **vpiClassDefn** when applied to a class obj handle returns the class defn with which the class obj was created. The difference between the two usages of **vpiClassDefn** can be seen in the example below:

```
class Packet;
    ...
endclass : Packet
class LinkedPacket extends Packet;
    ...
```

```
    endclass : LinkedPacket
LinkedPacket l = new;
Packet p = l;
```

In this example, the **vpiClassDefn** of variable p is Packet, but the **vpiClassDefn** of the class obj associated with variable p is "LinkedPacket".

d) **vpiClassDefn** shall return NULL for built-in classes.

e) The **vpiInterfaceDecl** iteration returns the virtual interfaces of the class object.

f) **vpi_get_value()** when applied to a class var reference handle shall provide the value of the handle to the class object or 0 if the class var is null.

g) **vpi_handle_by_name()** shall accept a full name to a nonstatic data member, even though it does not have a **vpiFullName** property. For example:

```
module top;

    class Packet ;
        integer Id ;
        ....
    endclass

    Packet p;
    c = p.Id;
    ....
```

vpi_handle_by_name() accepts "top.p.Id".

27.23 Constraint, constraint ordering, distribution
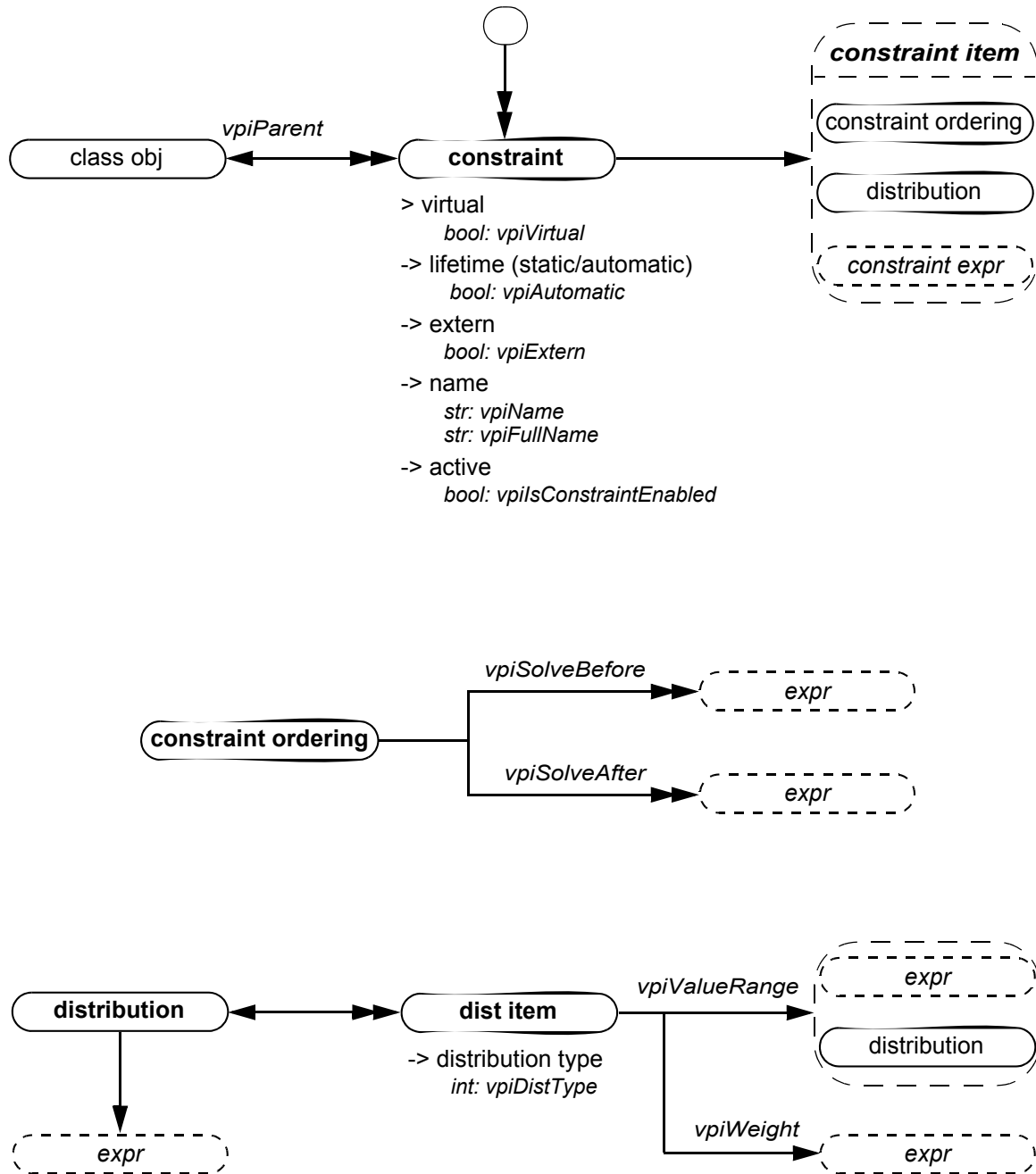
27.24 Constraint expression
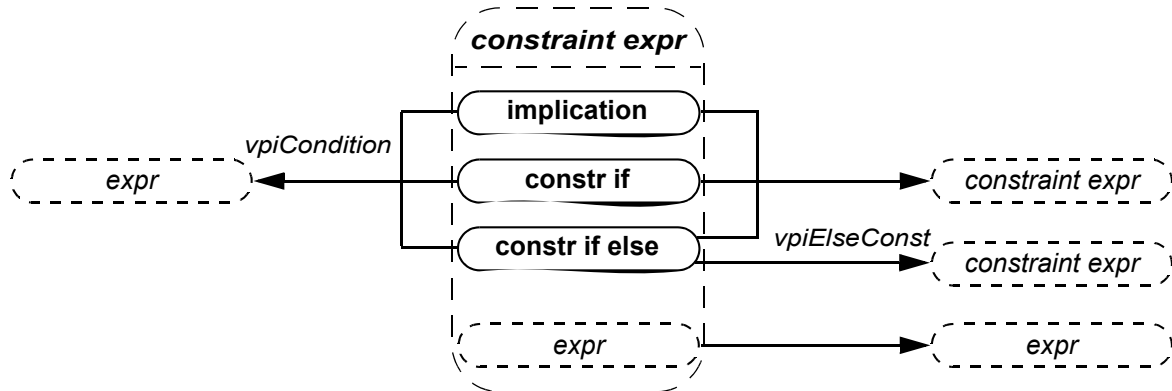
27.25 Module path, path term (supersedes 26.6.15 of IEEE Std 1364)
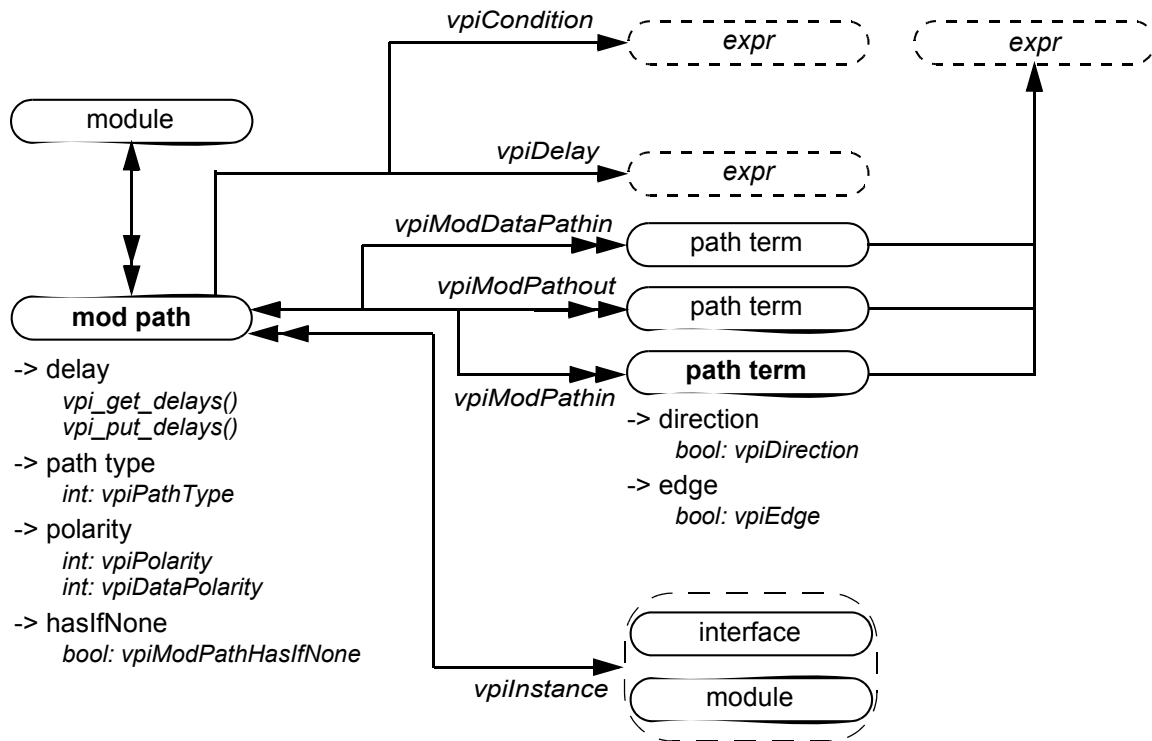
Details:

Specify blocks can occur in both modules and interfaces. For backwards compatibility, the **vpiModule** relation has been preserved; however, this relation shall return NULL for `specify` blocks in interfaces. For new code, it is recommended that the **vpiInstance** relation be used instead.

27.26 Task and function declaration (supersedes 26.6.18 of IEEE Std 1364)
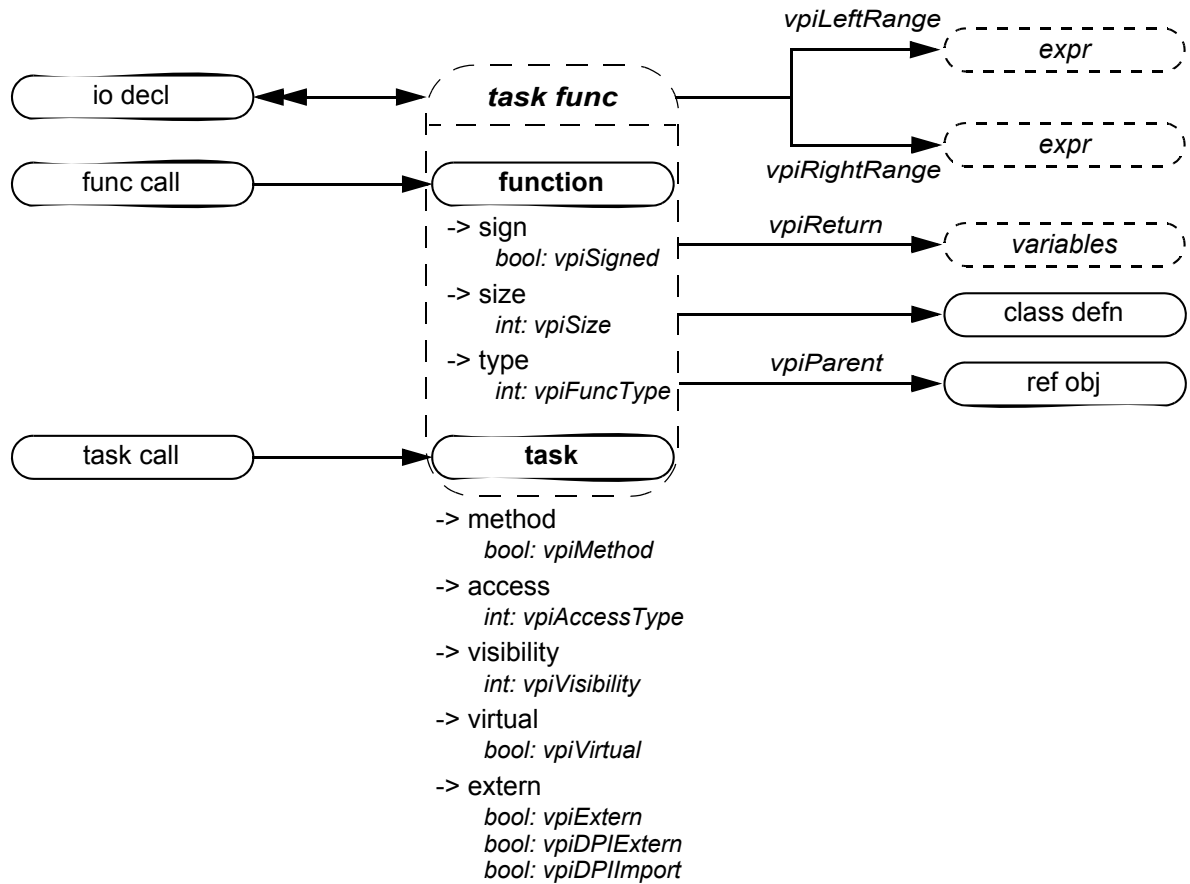
Details:

a) A Verilog HDL function shall contain an object with the same name, size, and type as the function. This object shall be used to capture the return value for this function.

b) For function where the return type is a user-defined type, **vpi_handle(vpiReturn,** function_handle**)** shall return the implicit variable handle representing the return of the function from which the user can get the details of that user-defined type.

c) **vpiReturn** shall always return a `var` object, even for simple returns.

d) **vpiVisibility** denotes the visibility (`local`, `protected`, or default) of a task or function that is a class member (a method). **vpiVisibility** shall return **vpiPublicVis** for a class member that is not local or protected or for a task or function that is not a class member.

e) **vpiFullName** of a task or function declared inside a package or class defn shall begin with the full name of the package or class defn followed by "`::`" and immediately followed with the name of the task or function.

27.27 Task and function call (supersedes 26.6.19 of IEEE Std 1364)
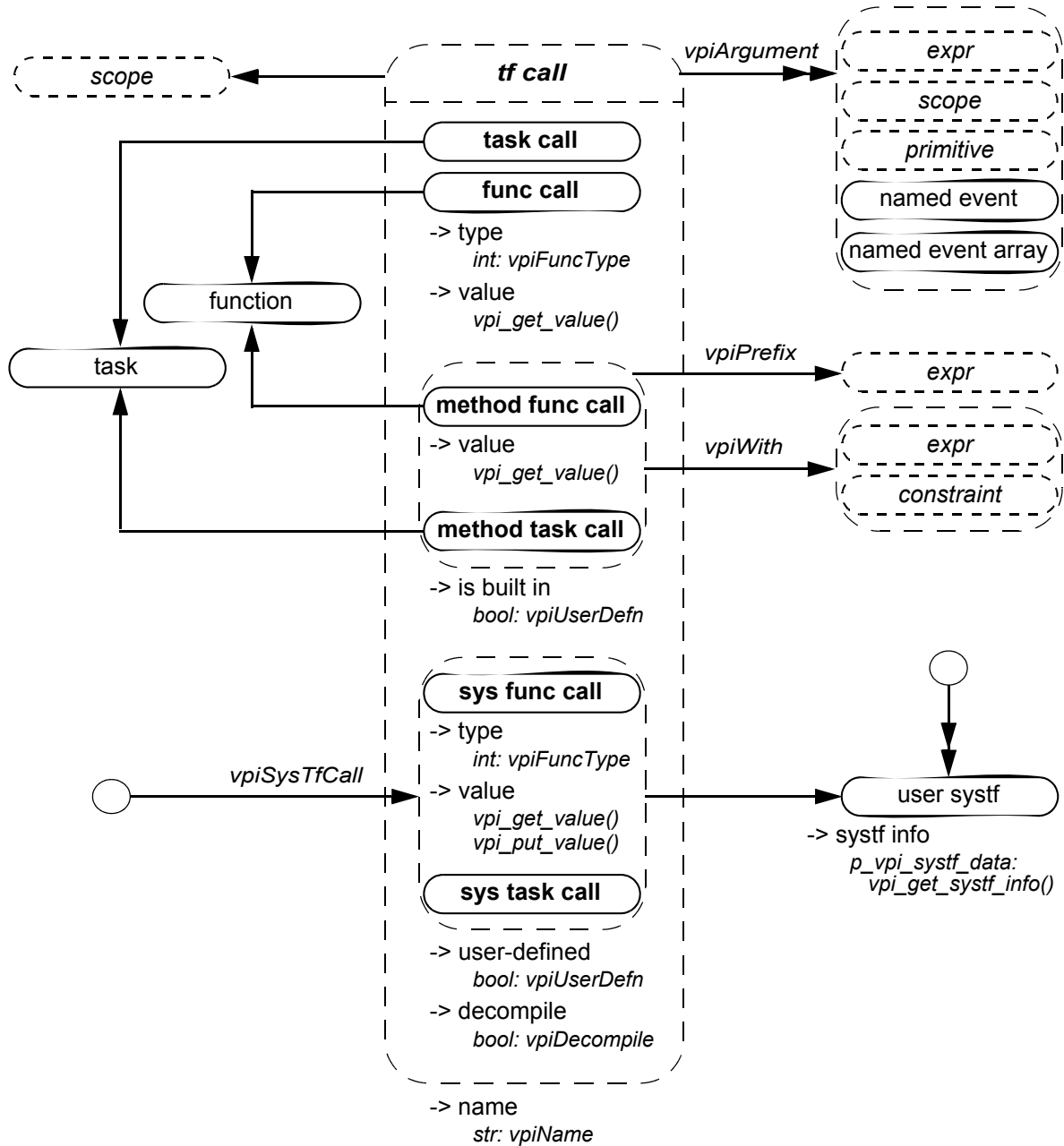
Details:

a) The **vpiWith** relation is only available for randomize methods (see 13.6) and for array locator methods (see 5.15.1).

b) For methods (method func call, method task call), the **vpiPrefix** relation shall return the object to which the method is being applied. For example, for the class method invocation

```
packet.send();
```

the prefix for the "send" method is the class var "packet".

c)   The system task or function that invoked an application shall be accessed with **vpi_handle (vpiSysTfCall,** NULL**)**.

d)   **vpi_get_value()** shall return the current value of the system function.

e)   If the **vpiUserDefn** property of a system task or function call is true, then the properties of the corresponding systf object shall be obtained via **vpi_get_systf_info()**.

f)   All user-defined system tasks or functions shall be retrieved using **vpi_iterate()**, with **vpiUserSystf** as the type argument, and a NULL reference argument.

g)   Arguments to PLI tasks or functions are not evaluated until an application requests their value. Effectively, the value of any argument is not known until the application asks for it. When an argument is an HDL or system function call, the function cannot be evaluated until the application asks for its value. If the application never asks for the value of the function, it is never evaluated. If the application has a handle to an HDL or system function, it may ask for its value at any time in the simulation. When this happens, the function is called and evaluated at this time.

h)   A null argument is an expression with a **vpiType** of **vpiOperation** and a **vpiOpType** of **vpiNullOp**.

i)   The property **vpiDecompile** shall return a string with a functionally equivalent system task or function call to what was in the original HDL. The arguments shall be decompiled using the same manner as any expression is decompiled. See 27.39 for a description of expression decompilation.

j)   System task and function calls that are protected shall allow iteration over the **vpiArgument** relationship.

27.28 Frames (supersedes 26.6.20 of IEEE Std 1364)
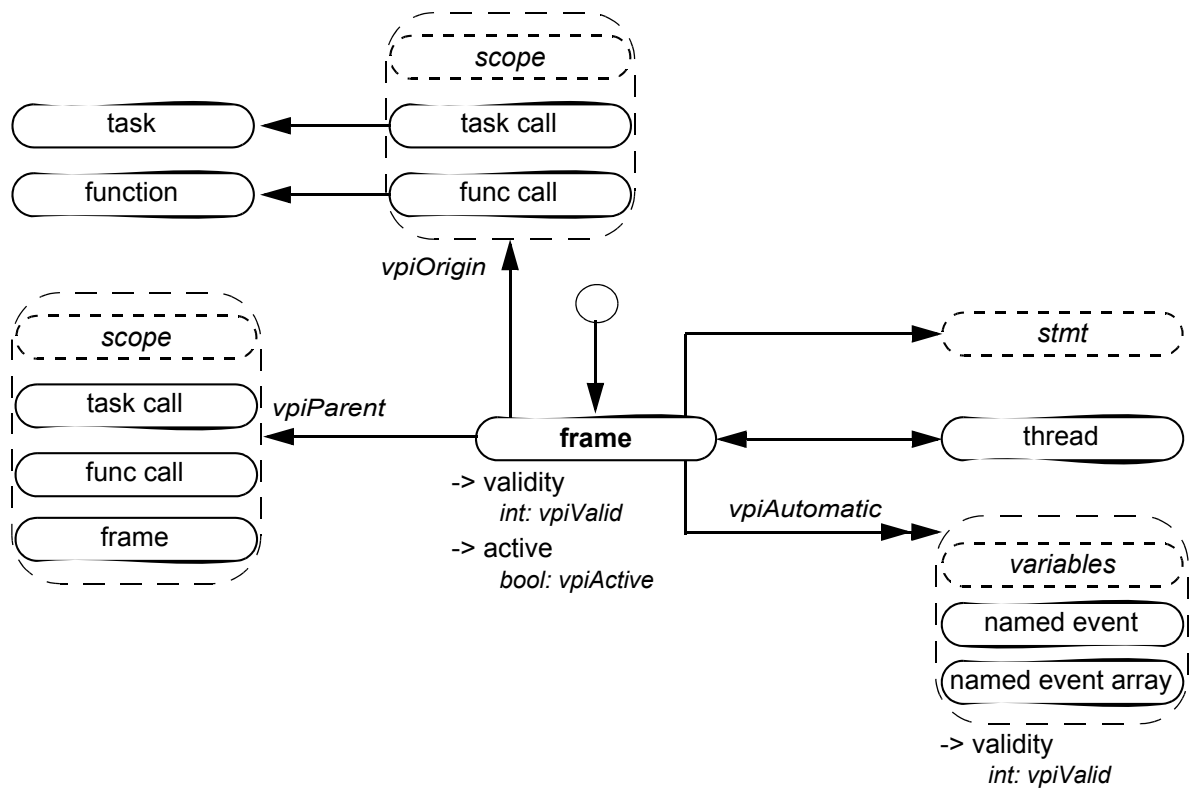
Details:

a)   Frames correspond to the set of automatic variables declared in a given task or function.

b) The following callbacks shall be supported on frames:

— **cbStartOfFrame** triggers whenever any frame is executed.

— **cbEndOfFrame** triggers when a particular frame is deleted after all storage is deleted.

c) It shall be illegal to place value change callbacks on automatic variables.

d) It shall be illegal to put a value with a delay on automatic variables.

e) There is at most only one active frame at any time in a given thread. To get a handle to the currently active frame, use **vpi_handle(vpiFrame,** NULL**)**. The frame to stmt transition shall return the currently active statement within the frame.

f) Frame handles must be freed using **vpi_free_object()** once the application no longer needs the handle. If the handle is not freed, it shall continue to exist, even after the frame has completed execution.

g) The frame object model is not backwards compatible with IEEE Std 1364.

27.29 Threads
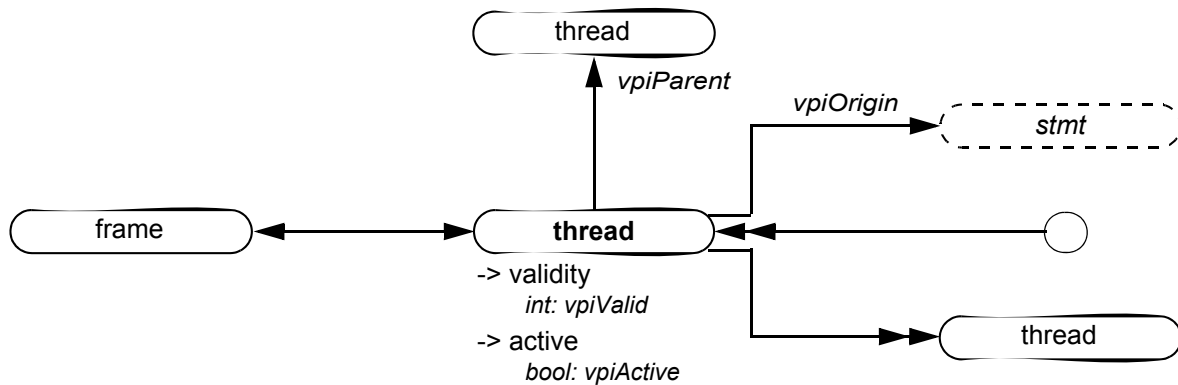
Details:

a) A thread is a Verilog process such as an **always** block or a branch of a **fork** construct. As a thread works its way down a call chain of tasks and/or functions, a new frame is activated as each new task or function is entered.

b) The following callbacks shall be supported on threads:

— **cbStartOfThread** triggers whenever any thread is created.

— **cbEndOfThread** triggers when a particular thread is deleted after storage is deleted.

— **cbEnterThread** triggers whenever a particular thread resumes execution.

27.30 Clocking block
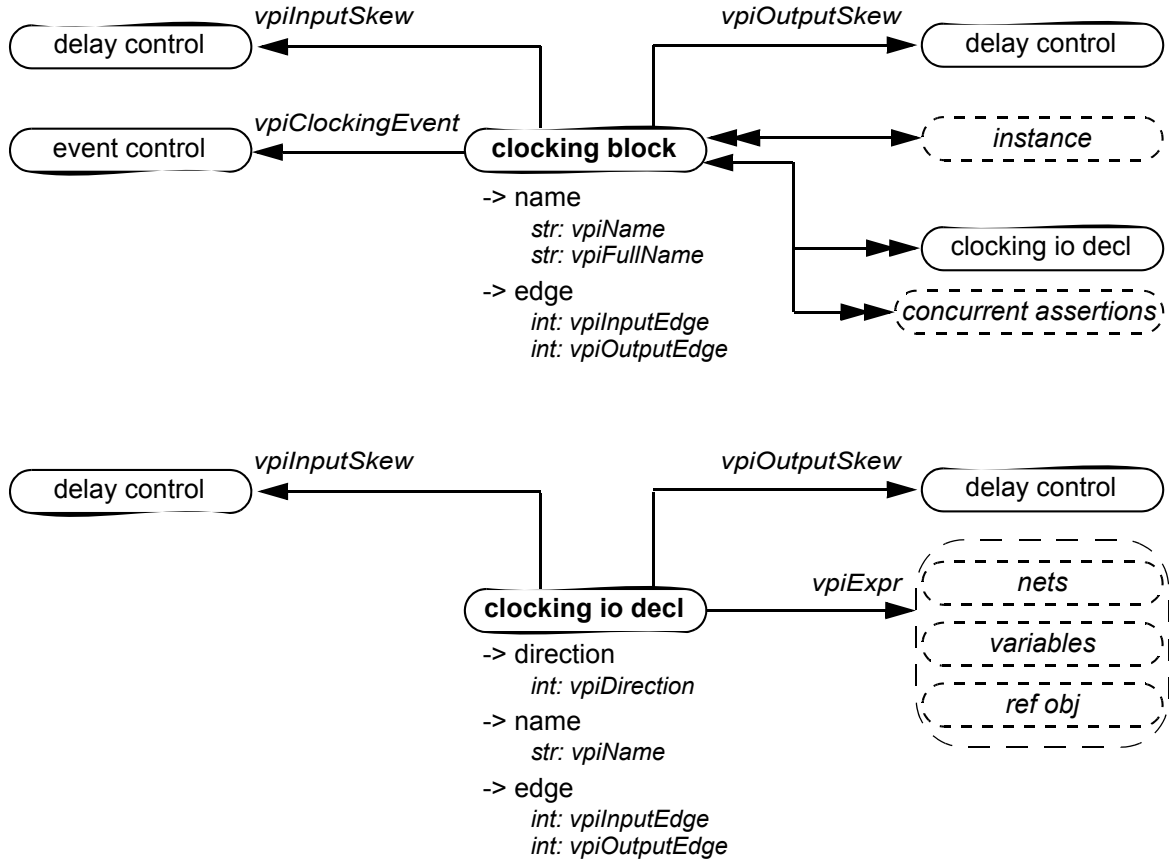
Details:

a)   The methods **vpiInputSkew** and **vpiOutputSkew** and properties **vpiInputEdge** and **vpiOutput-Edge** on the `clocking` block apply to the default constructs. The same methods and properties on the clocking io decl apply to the clocking io decl itself.

b)   **vpiExpr** shall return the expression or ref obj referenced by the clocking io decl. Consider `input enable = top.mem1.enable`. Here, "enable" is represented by a clocking io decl, and the **vpiExpr** relation returns a handle to "`top.mem1.enable`."

27.31 Assertion

```
                                    ◯
                                    │
                                    ▼
   ┌ ─ ─ ─ ─ ─ ─ ┐      ╭─────────────────╮              ╭──────────────────╮
   ╎  instance   ╎ ◄────┤    assertion    ├─────────────►│  clocking block  │
   └ ─ ─ ─ ─ ─ ─ ┘      ╎ ╭─────────────╮ ╎              ╰──────────────────╯
                        ╎ │ sequence inst│╎
                        ╎ ╰─────────────╯ ╎
                        ╎ ╭─────────────╮ ╎
                        ╎ │   assert     │╎
                        ╎ ╰─────────────╯ ╎
                        ╎ ╭─────────────╮ ╎
                        ╎ │   assume     │╎
                        ╎ ╰─────────────╯ ╎
                        ╎ ╭─────────────╮ ╎
                        ╎ │    cover     │╎
                        ╎ ╰─────────────╯ ╎
                        ╎ ╭─────────────╮ ╎
                        ╎ │ property inst│╎
                        ╎ ╰─────────────╯ ╎
                        ╎ ╭─────────────╮ ╎
                        ╎ │immediate assert│╎
                        ╎ ╰─────────────╯ ╎
                        └ ─ ─ ─ ─ ─ ─ ─ ─ ┘
```

-> assertion type
    int: vpiAssertType

-> location
    str: vpiFile
    int: vpiStartLine
    int: vpiColumn
    int: vpiEndLine
    int: vpiEndColumn

-> assertion name
    str: vpiName
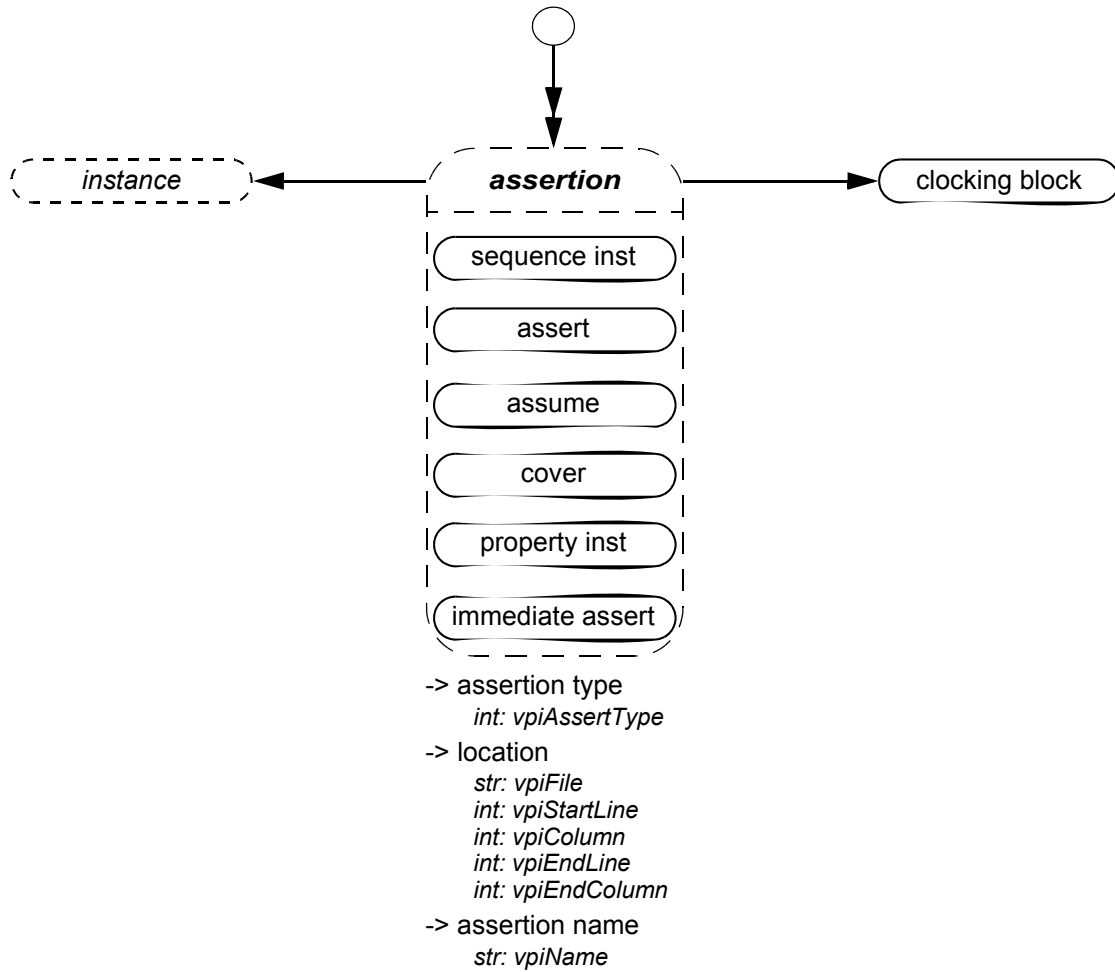
27.32 Concurrent assertions
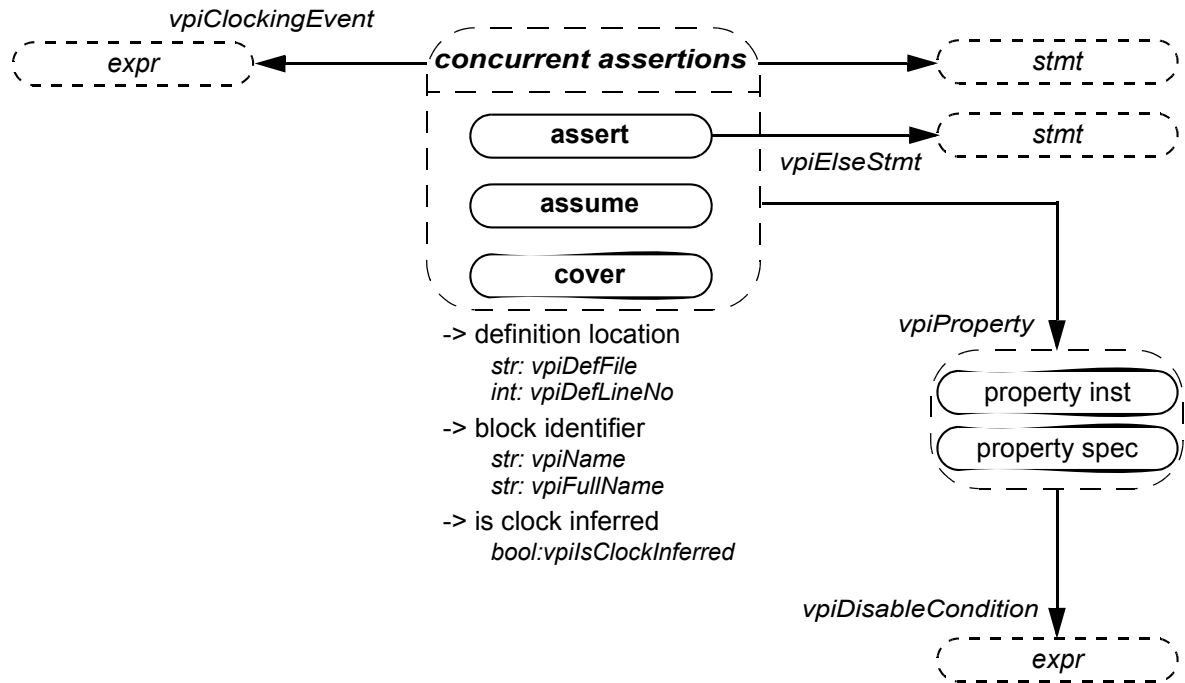
Details:

Clocking event is always the actual clocking event on which the assertion is being evaluated, regardless of whether this is explicit or implicit (inferred).

27.33 Property declaration
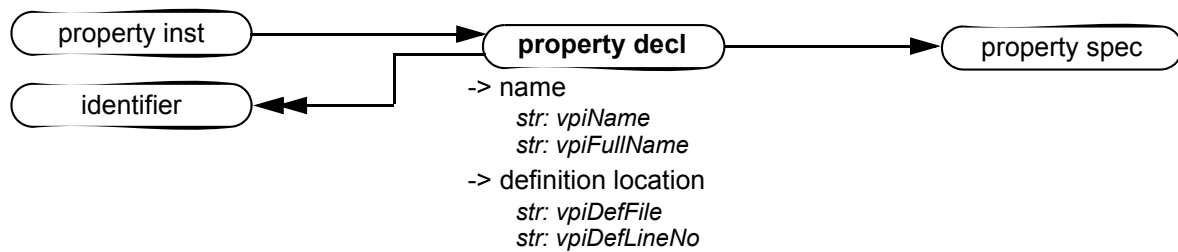
27.34 Property specification
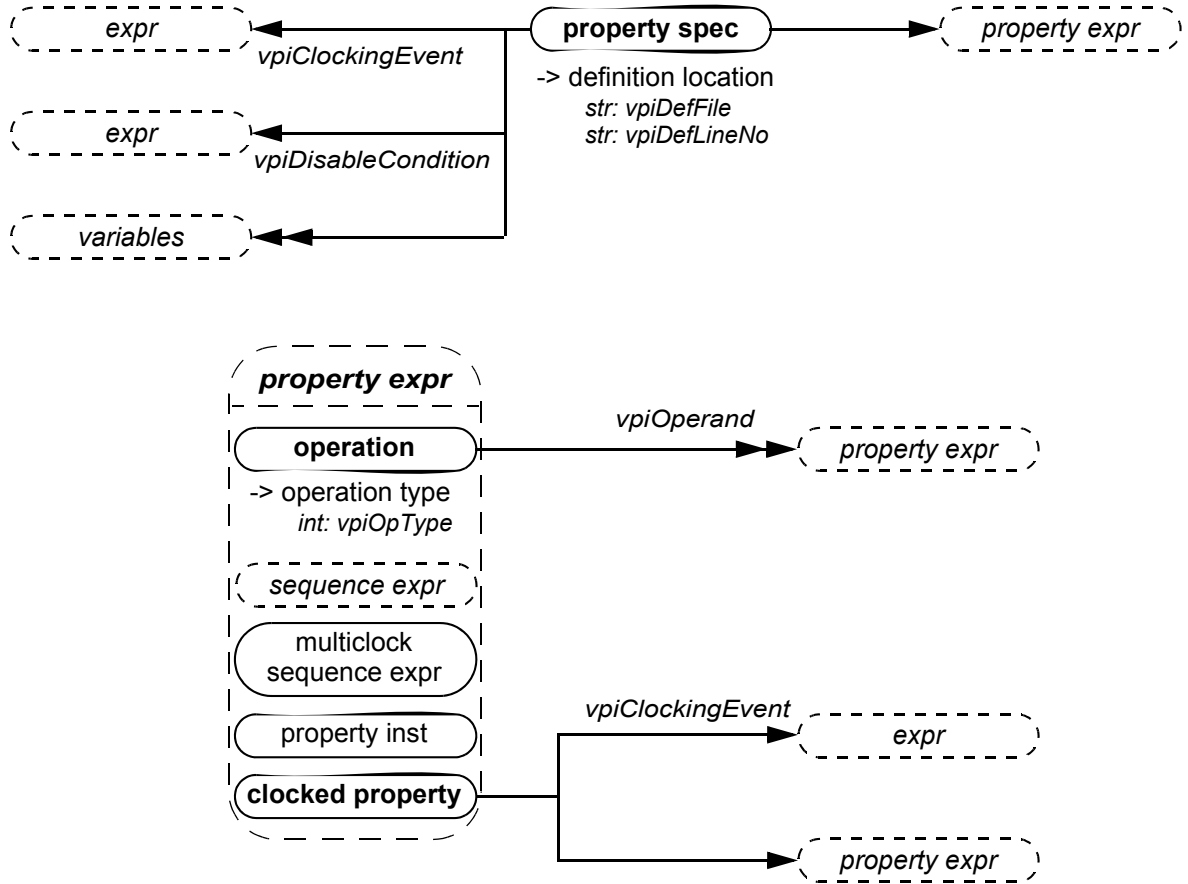
Details:

a) Variables are declarations of property variables. The value of these variables cannot be accessed.

b) Within the context of a property expr, **vpiOpType** can be any one of **vpiNotOp**, **vpiOverlapImplyOp**, **vpiNonOverlapImplyOp**, **vpiCompAndOp**, **vpiCompOrOp**, **vpiIfOp** or **vpiIfElseOp**. Operands to these operations shall be provided in the same order as show in the BNF.

27.35 Sequence declaration
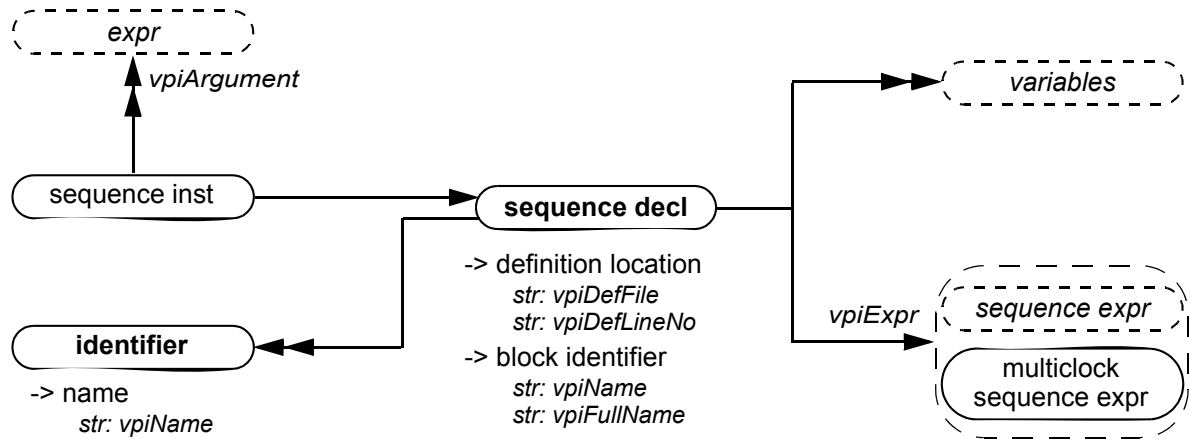
Details:

The **vpiArgument** iterator shall return the sequence instance arguments in the order that the formals for the sequence are declared so that the correspondence between each argument and its respective formal can be made. If a formal has a default value, that value shall appear as the argument should the instantiation not provide a value for that argument.

27.36 Sequence expression
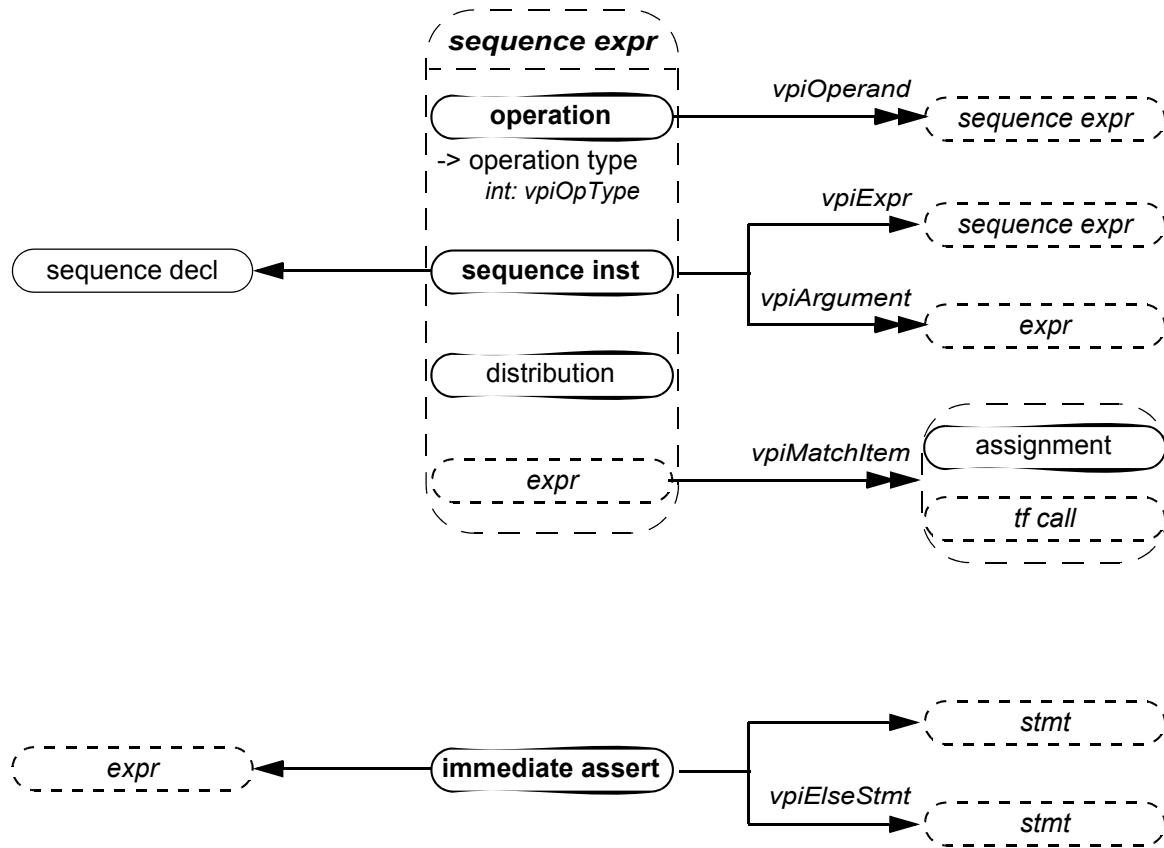
Details:

a) Within a sequence expression, **vpiOpType** can be any one of **vpiCompAndOp**, **vpiIntersectOp**, **vpiCompOr**, **vpiFirstMatchOp**, **vpiThroughoutOp**, **vpiWithinOp**, **vpiUnaryCycleDelayOp**, **vpiCycleDelayOp**, **vpiRepeatOp**, **vpiConsecutiveRepeatOp**, or **vpiGotoRepeatOp**.

b) For operations, the operands shall be provided in the same order as the operands appear in BNF, with the following exceptions:

— **vpiUnaryCycleDelayOp**: Arguments shall be sequence, left range, and right range. Right range shall only be given if different from left range.

— **vpiCycleDelayOp**: Arguments shall be LHS sequence, rhs sequence, left range, and right range. Right range shall only be provided if different from left range.

— All the repeat operators: The first argument shall be the sequence being repeated, and the next argument shall be the left repeat bound, followed by the right repeat bound. The right repeat bound shall only be provided if different from left repeat bound.

and, intersect, or,
first_match,
throughout, within,
##,
[*], [=], [->]

27.37 Multiclock sequence expression
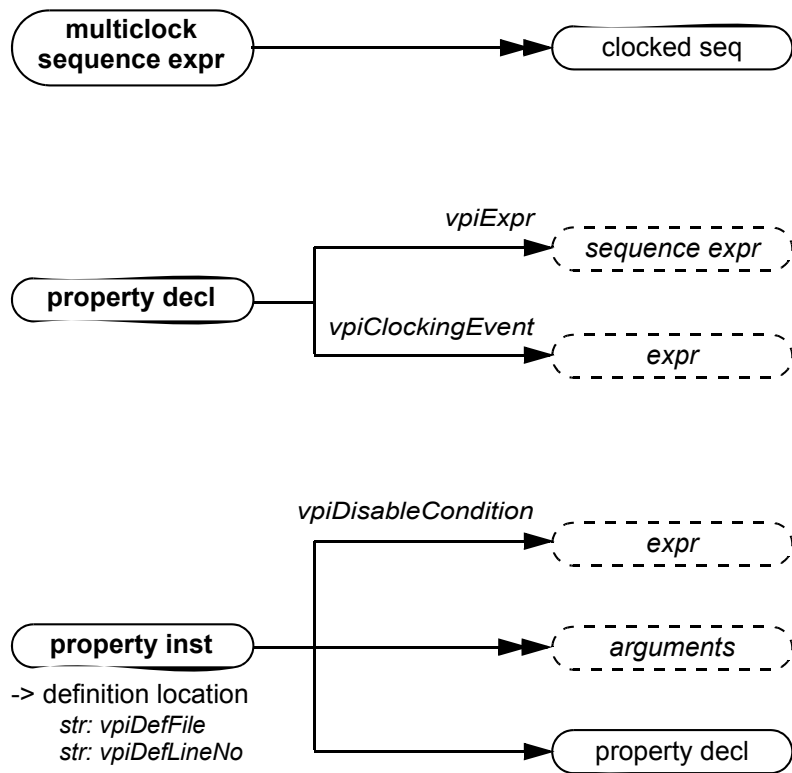
27.38 Simple expressions (supersedes 26.6.25 of IEEE Std 1364)
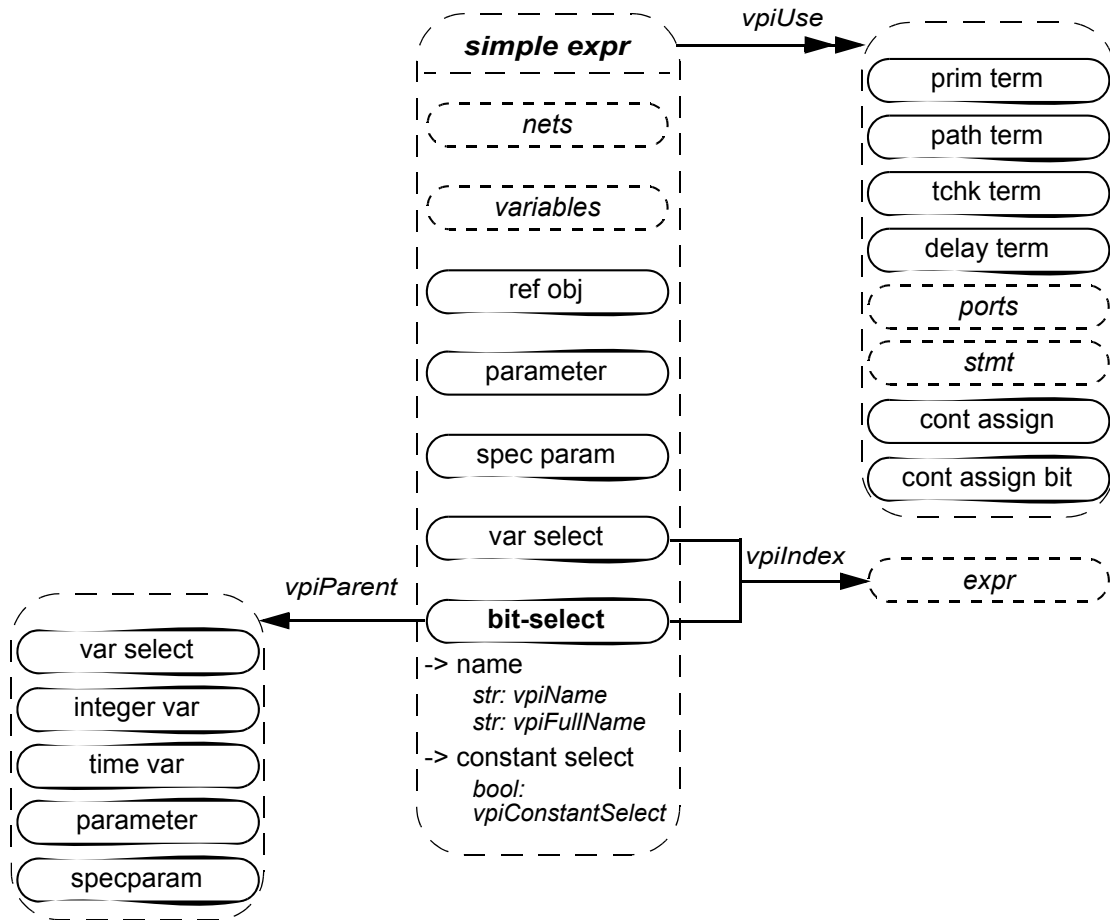
Details:

a) For vectors, the **vpiUse** relationship shall access any use of the vector or of the part-selects or bit-selects of the vector.

b) For bit-selects, the **vpiUse** relationship shall access any specific use of that bit, any use of the parent vector, and any part-select that contains that bit.

27.39 Expressions (supersedes 26.6.26 of IEEE Std 1364)
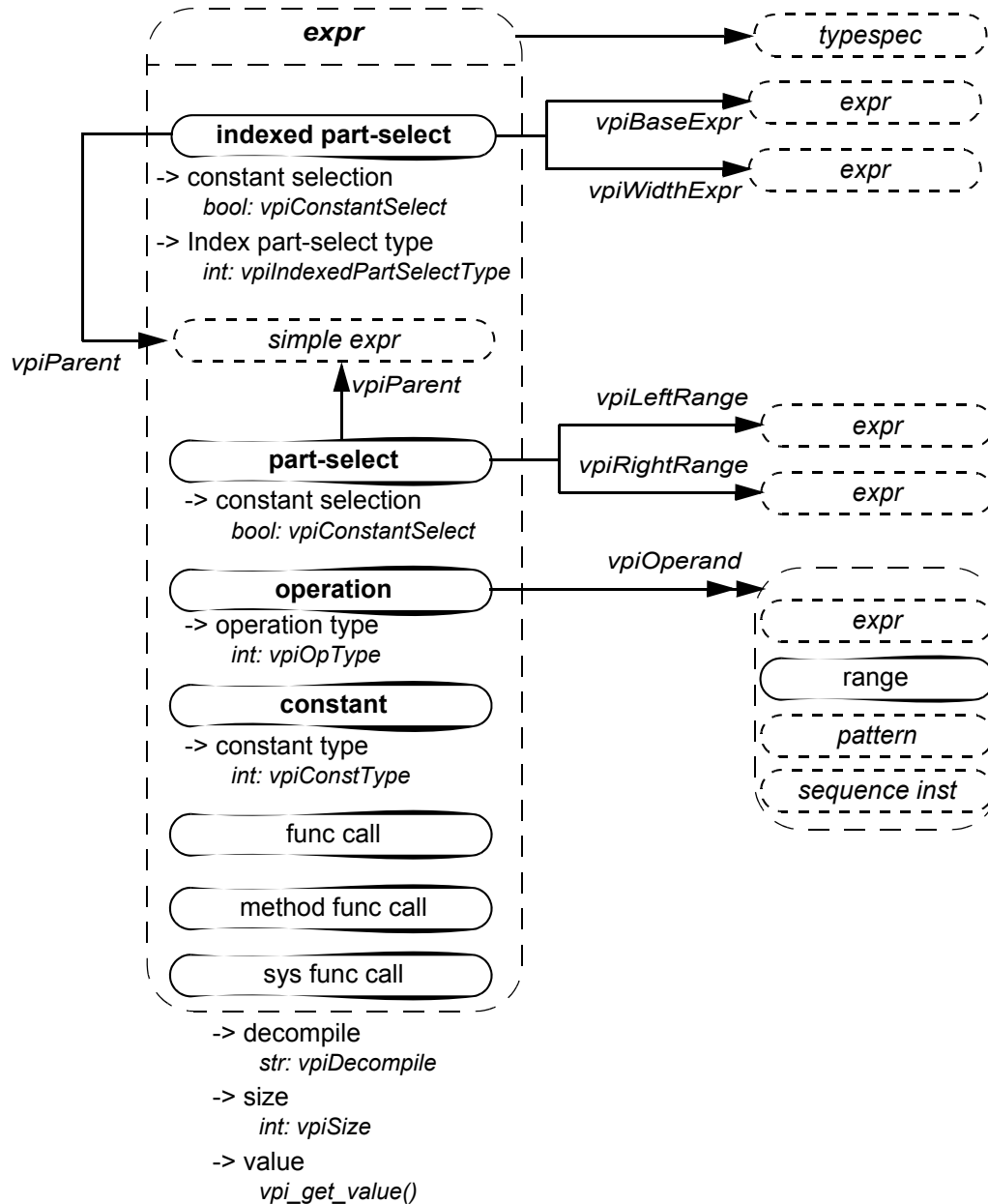
Details:

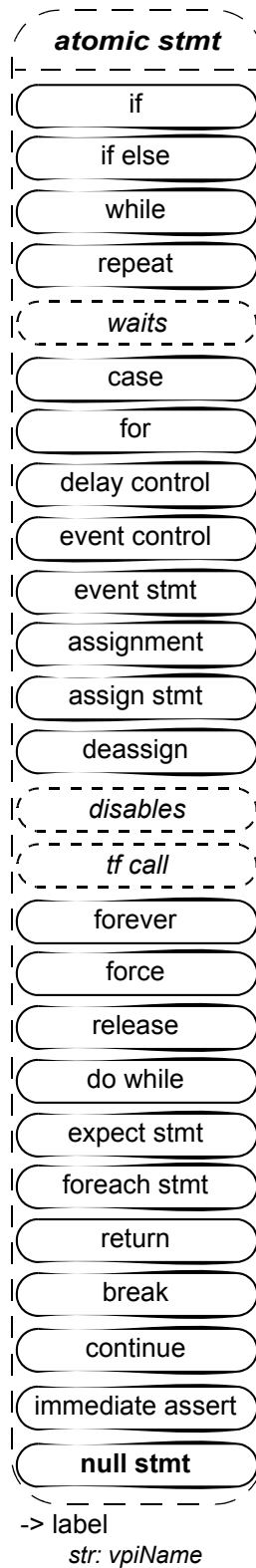a)   For an operator whose type is **vpiMultiConcatOp**, the first operand shall be the multiplier expression. The remaining operands shall be the expressions within the concatenation.

b)   The property **vpiDecompile** shall return a string with a functionally equivalent expression to the original expression within the HDL. Parentheses shall be added only to preserve precedence. Each operand and operator shall be separated by a single space character. No additional white space shall be added due to parentheses.

c)   The cast operation, for which **vpiOpType** returns **vpiCastOp**, is represented as a unary operation, with its sole argument being the expression being cast and the typespec of the cast expression being the type to which the argument is being cast.

d)   The constant type **vpiUnboundedConst** represents the **$** value used in assertion ranges.

e)   The one-to-one relation to typespec must always be available for **vpiCastOp** operations, for simple expressions, and for **vpiAssignmentPatternOp** and **vpiMultiAssignmentPatternOp** expressions when the curly braces of the assignment pattern are prefixed by a data type name to form an assignment pattern expression. For other expressions, it is implementation dependent whether there is any associated typespec.

f)   For an operation of type **vpiAssignmentPatternOp**, the operand iteration shall return the expressions as if the assignment pattern were written with the positional notation. Nesting of assignment patterns shall be preserved.

Example 1:

```
struct {
   int A;
   struct {
     logic B;
     real C;
   } BC1, BC2;
}ABC = '{BC1: '{1'b1, 1.0}, int: 0, BC2: '{default: 0}};
```

The assignment pattern that initializes the struct variable ABC uses member, type, and default keys. The **vpiOperand** traversal would represent this assignment pattern as follows:

```
'{0, '{1'b1, 1.0}, '{0, 0}}
```

or some other equivalent positional assignment pattern.

Example 2:

```
logic [2:0] varr [0:3] = '{3: 3'b1, default: 3'b0};
```

The assignment pattern that initializes the array variable varr uses index and default keys. The **vpiOperand** traversal would represent this assignment pattern as follows:

```
'{3'b0, 3'b0, 3'b0, 3'b1}
```

g)   For an operator whose type is **vpiMultiAssignmentPatternOp**, the first operand shall be the multiplier expression. The remaining operands shall be the expressions within the assignment pattern.

Example:

```
bit unpackedbits [1:0];
initial unpackedbits = '{2 {y}} ; // same as '{y, y}
```

For the assignment pattern '{2{y}}, the **vpiOpType** property shall return **vpiMultiAssignment-PatternOp**, and the first operand shall be the constant 2. The next operand shall represent the expression y.

h)   Expressions that are protected shall permit access to the **vpiSize** property.

## 27.40 Atomic statement (supersedes atomic stmt in 26.6.27 of IEEE Std 1364)


```
atomic stmt
    if
    if else
    while
    repeat
    waits
    case
    for
    delay control
    event control
    event stmt
    assignment
    assign stmt
    deassign
    disables
    tf call
    forever
    force
    release
    do while
    expect stmt
    foreach stmt
    return
    break
    continue
    immediate assert
    null stmt
-> label
  str: vpiName
```

Details:

The **vpiName** property shall provide the statement label if one was given; otherwise, the name is NULL.

27.41 Event statement (supersedes event stmt in 26.6.27 of IEEE Std 1364)

27.42 Process (supersedes process in 26.6.27 of IEEE Std 1364)
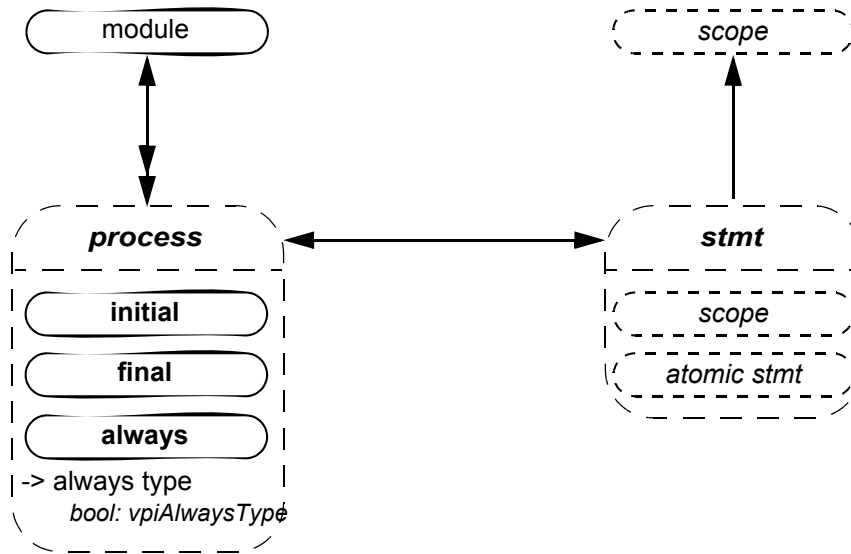
Details:

vpiAlwaysType can be one of **vpiAlways**, **vpiAlwaysComb**, **vpiAlwaysFF**, or **vpiAlwaysLatch**.

27.43 Assignment (supersedes 26.6.28 of IEEE Std 1364)
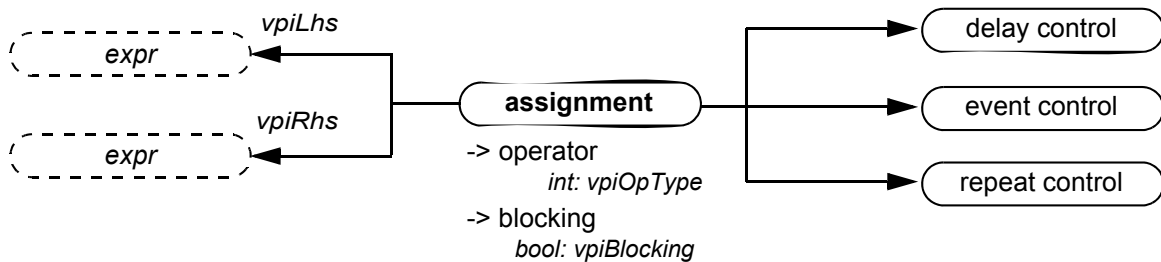
Details:

vpiOpType shall return **vpiAssignmentOp** for normal assignments (both blocking '=' and nonblocking '<=') or the **vpiOpType** of the operators described in 8.3.

For example, the assignment

```
a += 2;
```

shall return **vpiAddOp** for the **vpiOpType** property.

27.44 Event control (supersedes 26.6.30 of IEEE Std 1364)
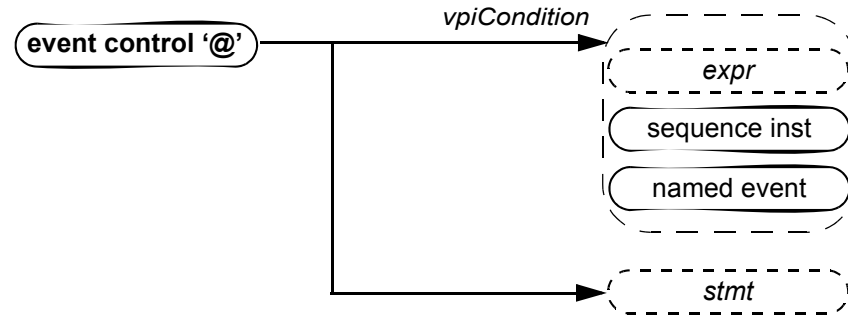
Details:

For event control associated with assignment, the statement shall always be NULL.

27.45 Waits (supersedes wait in 26.6.32 of IEEE Std 1364)
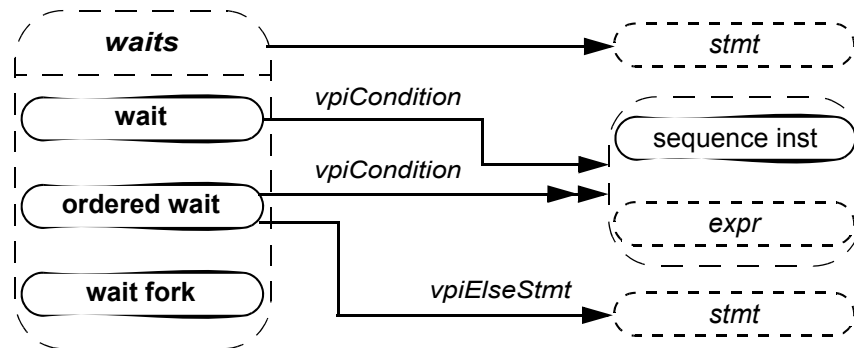
27.46 If, if-else (supersedes 26.6.35 of IEEE Std 1364)

-> qualifier
    int: vpiQualifer
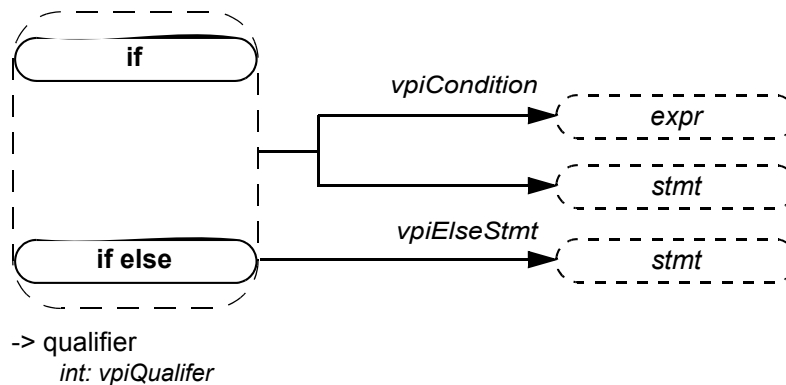
27.47 Case, pattern (supersedes 26.6.36 of IEEE Std 1364)
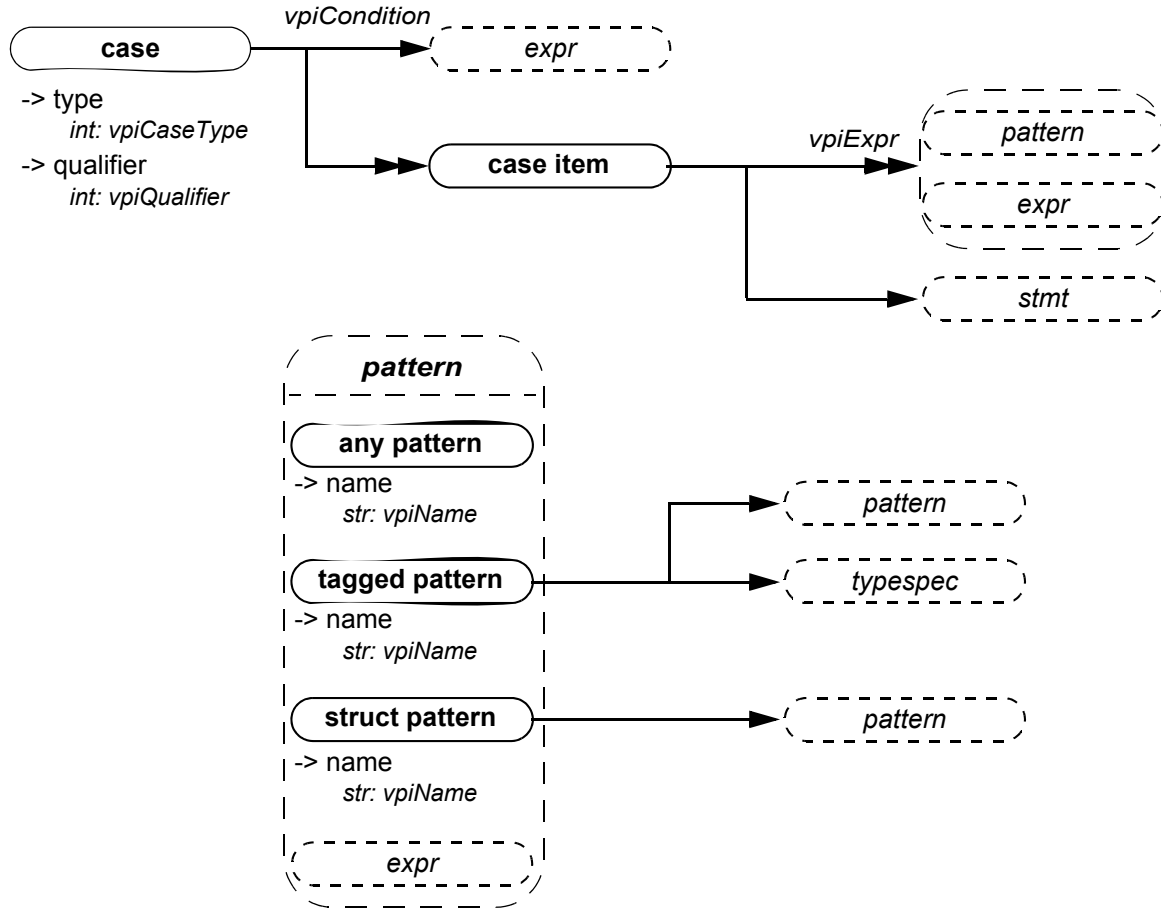
Details:

a) The case item shall group all case conditions that branch to the same statement.

b) **vpi_iterate()** shall return NULL for the default case item because there is no expression with the default case.

27.48 Expect
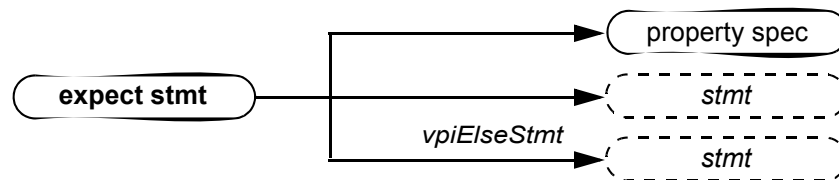
27.49 For (supersedes 26.6.33 of IEEE Std 1364)
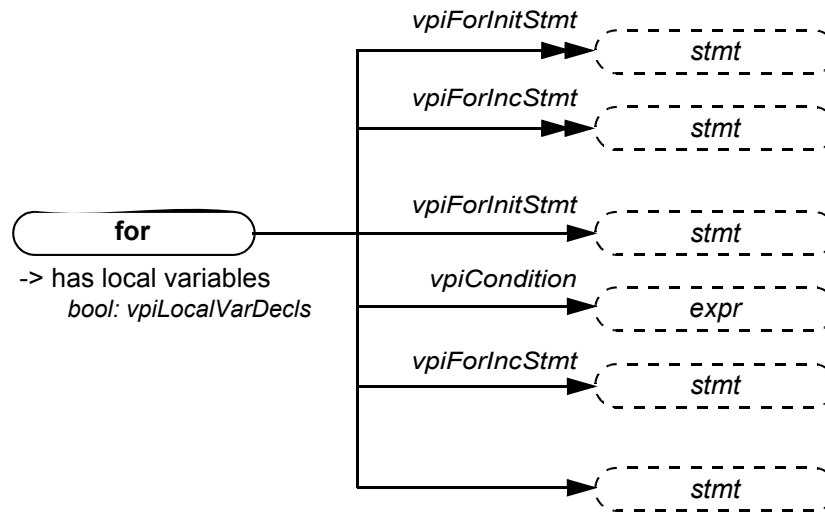
27.50 Do-while, foreach
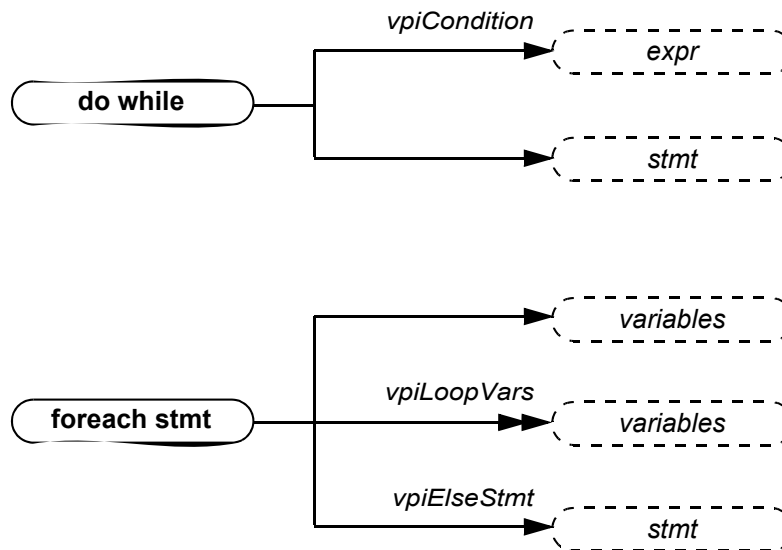
Details:

The variable obtained via the **vpiVariable** relation from a foreach stmt shall always be of type **vpiArrayVar**.

27.51 Alias statement
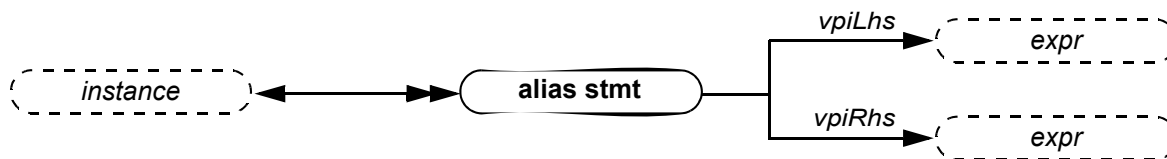
Examples:

```
    alias a=b=c=d;
```

results in three aliases:

```
        alias a=d;
        alias b=d;
        alias c=d;
```
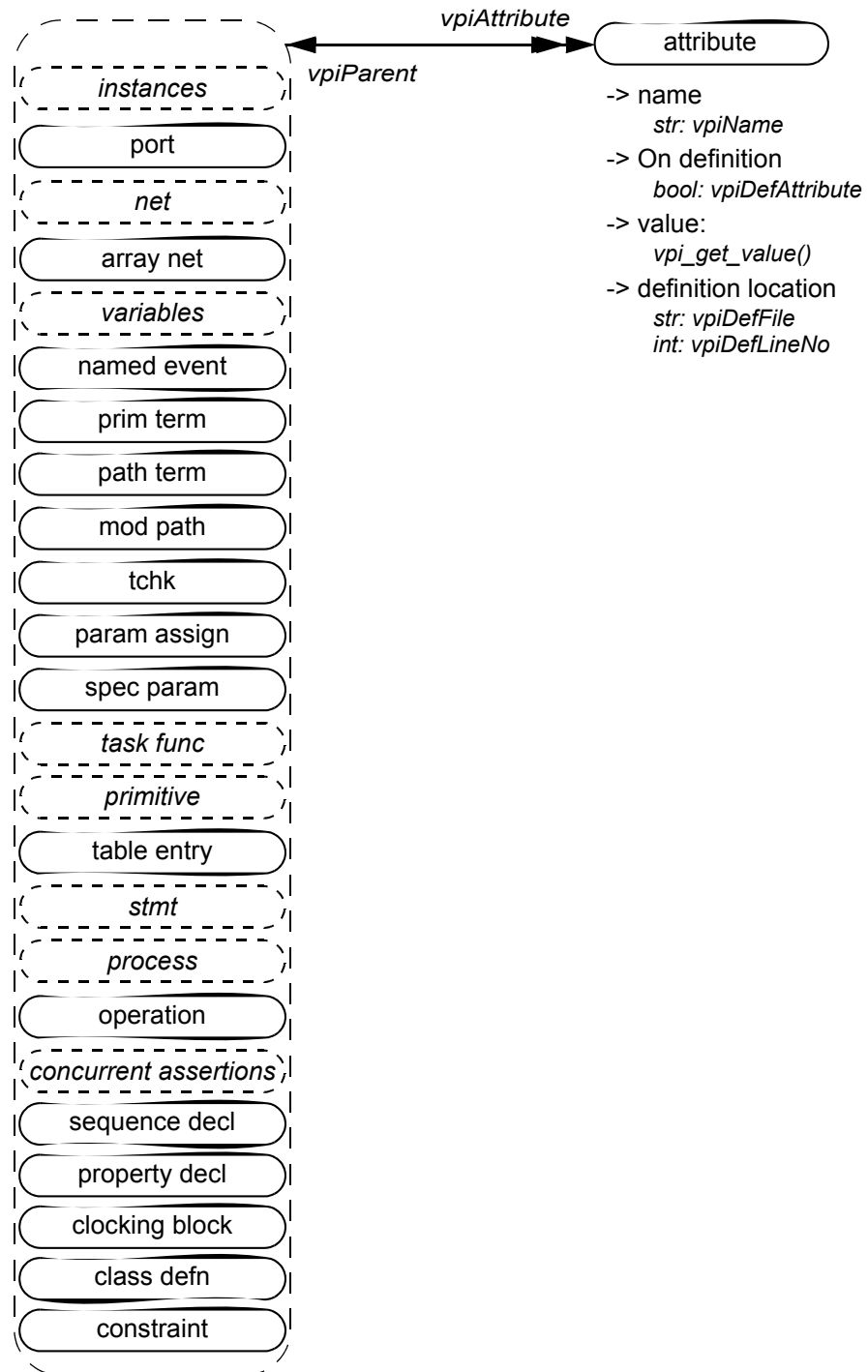
d is the Rhs for all.
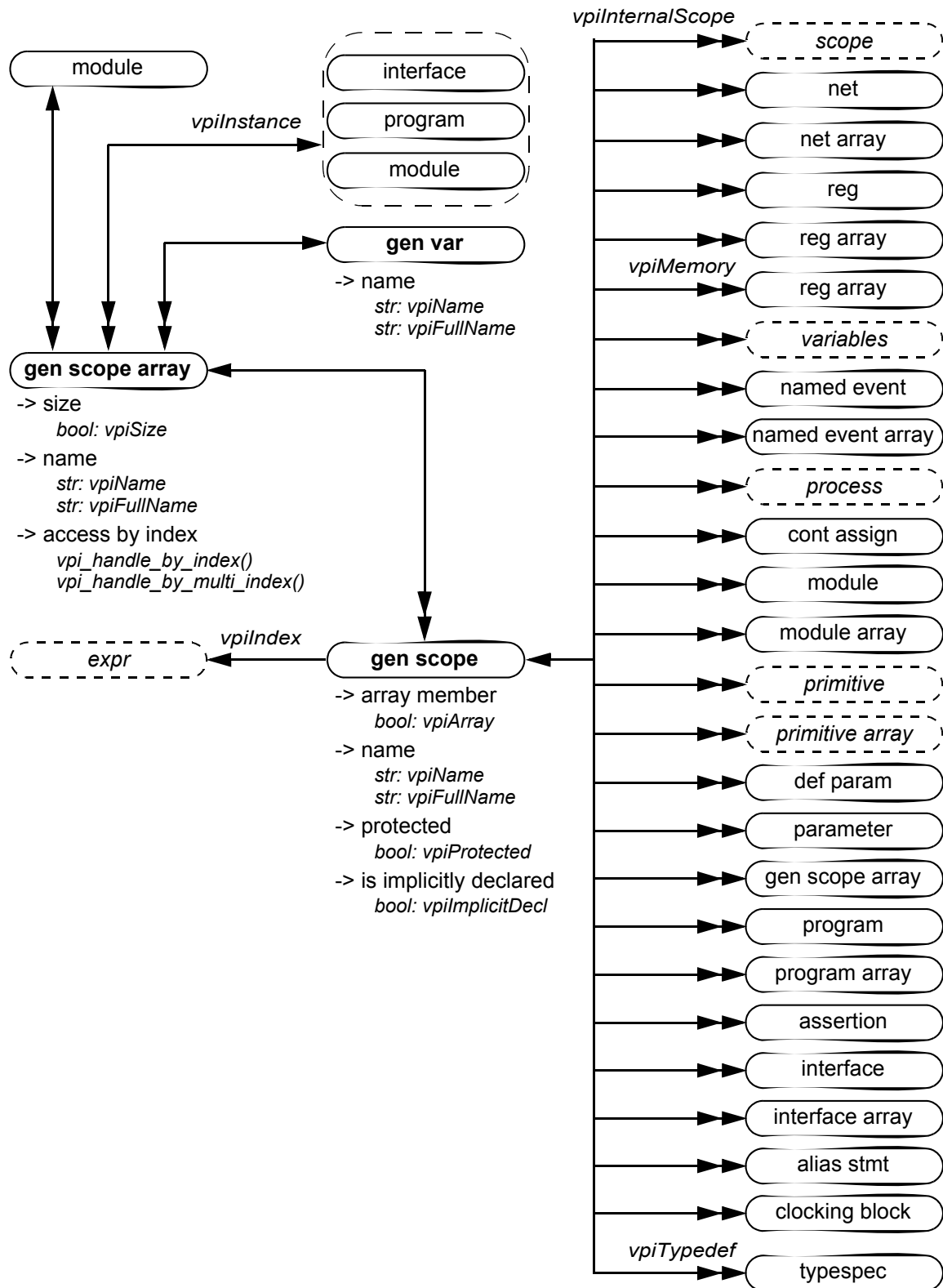
27.52 Disables (supersedes 26.6.38 of IEEE Std 1364)
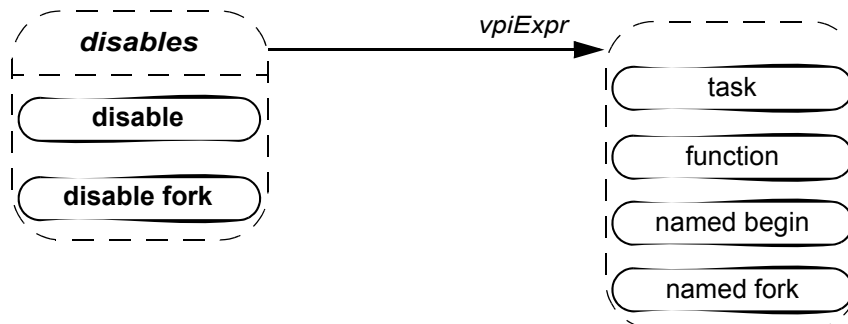
27.53 Return statement
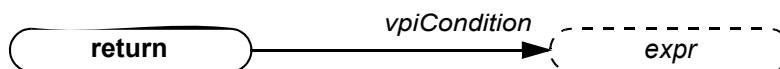
27.54 Attribute (supersedes 26.6.42 of IEEE Std 1364)

27.55 Generates (supersedes 26.6.44 of IEEE Std 1364)

Details:

a) The size for a gen scope array is the number of elements in the array.

b)   For unnamed generates, an implicit scope shall be created. Its **vpiImplicitDecl** property shall return `TRUE`.

c)   References to gen vars within the gen scope shall be treated as local parameters parameter.

d)   Parameters within the gen scope must be local.

28. SystemVerilog assertion API

28.1 Requirements

This clause defines the assertion API in SystemVerilog.

SystemVerilog provides assertion capabilities to enable the following:
— A user's C code to react to assertion events
— Third-party assertion "waveform" dumping tools to be written
— Third-party assertion coverage tools to be written
— Third-party assertion debug tools to be written

28.2 Static information

This subclause defines how to obtain assertion handles and other static assertion information.

28.2.1 Obtaining assertion handles

SystemVerilog extends the VPI navigation model to encompass assertions, properties, and sequences. It also enhances the instance iterator model with direct access to assertions, properties, and sequences.

The following steps highlight how to obtain the assertion handles for named assertions through direct access.

a) Iterate all assertions in the design: use a NULL reference handle (ref) to vpi_iterate(). For example:

```
itr = vpi_iterate(vpiAssertion, NULL);
while (assertion = vpi_scan(itr)) {
   /* process assertion */
}
```

b) Iterate all assertions in an instance: pass the appropriate instance handle as a reference handle to vpi_iterate(). For example:

```
itr = vpi_iterate(vpiAssertion, instanceHandle);
while (assertion = vpi_scan(itr)) {
   /* process assertion */
}
```

c) Obtain the assertion by name: extend vpi_handle_by_name to also search for assertion names in the appropriate scope(s). For example:

```
vpiHandle = vpi_handle_by_name(assertName, scope)
```

d) To obtain an assertion of a specific type, e.g., cover assertions, the following approach should be used:

```
vpiHandle assertion;
itr = vpi_iterate(vpiAssertion, NULL);
while (assertion = vpi_scan(itr)) {
    if (vpi_get(vpiType, assertion) == vpiCover) {
```

```
                    /* process cover type assertion */
            }
        }
```

Details:

— As with all VPI handles, assertion handles are handles to a specific instance of a specific assertion.

— Unnamed assertions cannot be found by name.

28.2.2 Obtaining static assertion information

The following information about an assertion is considered to be static:

— Assertion name

— Instance in which the assertion occurs

— Module definition containing the assertion

— Assertion type

    — Sequence

    — Assert

    — Assume

    — Cover

    — Property

    — ImmediateAssert

— Assertion source information: the file, line, and column where the assertion is defined

— Assertion **clocking** block/expression

28.3 Dynamic information

This subclause defines how to place assertion system and assertion callbacks.

28.3.1 Placing assertion system callbacks

To place an assertion system callback, use `vpi_register_cb()`, setting the `cb_rtn` element to the function to be invoked and the reason element of the `s_cb_data` structure to one of the following values:

— `cbAssertionSysInitialized`. This callback occurs after the system has initialized. No assertion-specific actions can be performed until this callback completes. The assertion system can initialize before `cbStartOfSimulation` does or afterwards.

— `cbAssertionSysOn`. The assertion system has become active and starts processing assertion attempts. This always occur after `cbAssertionSysInitialized`. By default, the assertion system is "started" on simulation startup, but the user can delay this by using assertion system control actions.

— `cbAssertionSysOff`. The assertion system has been temporarily suspended. While stopped, no new assertion attempts are processed and no new assertion-related callbacks occur. Assertions already executing are not affected. The assertion system can be stopped and resumed an arbitrary number of times during a single simulation run.

— `cbAssertionSysKill`. The assertion system has been temporarily suspended. While suspended, no assertion attempts are processed, and no assertion-related callbacks occur. The assertion system can be suspended and resumed an arbitrary number of times during a single simulation run.

— `cbAssertionSysEnd`. This callback occurs when all assertions have completed and no new attempts shall start. Once this callback occurs, no more assertion-related callbacks shall occur, and

assertion-related actions shall have no further effect. This typically occurs after the end of
simulation.

— `cbAssertionSysReset`. This callback occurs when the assertion system is reset, e.g., due to a sys-
tem control action.

The callback routine invoked follows the normal VPI callback prototype and is passed an `s_cb_data` con-
taining the callback reason and any user data provided to the `vpi_register_cb()` call.

28.3.2 Placing assertions callbacks

To place an assertion callback, use `vpi_register_assertion_cb()`. The prototype is as follows:

```
/* typedef for vpi_register_assertion_cb callback function */
typedef  PLI_INT32 (vpi_assertion_callback_func)(
   PLI_INT32 reason,            /* callback reason */
   p_vpi_time cb_time,          /* callback time */
   vpiHandle assertion,         /* handle to assertion */
   p_vpi_attempt_info info,     /* attempt related information */
   PLI_BYTE8 *user_data         /* user data entered upon registration */
);

vpiHandle vpi_register_assertion_cb(
   vpiHandle assertion,         /* handle to assertion */
   PLI_INT32 reason,            /* reason for which callbacks needed */
   vpi_assertion_callback_func *cb_rtn,
   PLI_BYTE8 *user_data         /* user data to be supplied to cb */
);

typedef struct t_vpi_assertion_step_info {
   PLI_INT32 matched_expression_count;
   vpiHandle *matched_exprs;  /* array of expressions */
   PLI_INT32 stateFrom, stateTo;/* identify transition */
} s_vpi_assertion_step_info, *p_vpi_assertion_step_info;

typedef struct t_vpi_attempt_info {
   union {
      vpiHandle failExpr;
      p_vpi_assertion_step_info step;
   } detail;
   s_vpi_time attemptStartTime;  /* Time attempt triggered */
} s_vpi_attempt_info, *p_vpi_attempt_info;
```

where *reason* is any of the following.

— `cbAssertionStart`. An assertion attempt has started. For most assertions, one attempt starts each
and every clock tick.

— `cbAssertionSuccess`. An assertion attempt reaches a success state.

— `cbAssertionFailure`. An assertion attempt fails to reach a success state.

— `cbAssertionStepSuccess`. Progress one step an attempt. By default, step callbacks are not
enabled on any assertions; they are enabled on a per-assertion/per-attempt basis (see 28.4.2), rather
than on a per-assertion basis.

— `cbAssertionStepFailure`. Fail to progress by one step along an attempt. By default, step call-
backs are not enabled on any assertions; they are enabled on a per-assertion/per-attempt basis (see
28.4.2), rather than on a per-assertion basis.

— `cbAssertionDisable`. The assertion is disabled (e.g., as a result of a control action).

— `cbAssertionEnable`. The assertion is enabled.

— `cbAssertionReset`. The assertion is reset.

— `cbAssertionKill`. An attempt is killed (e.g., as a result of a control action).

These callbacks are specific to a given assertion; placing such a callback on one assertion does not cause the callback to trigger on an event occurring on a different assertion. If the callback is successfully placed, a handle to the callback is returned. This handle can be used to remove the callback via `vpi_remove_cb()`. If there were errors on placing the callback, a `NULL` handle is returned. As with all other calls, invoking this function with invalid arguments has unpredictable effects.

Once the callback is placed, the user-supplied function shall be called each time the specified event occurs on the given assertion. The callback shall continue to be called whenever the event occurs until the callback is removed.

The callback function shall be supplied the following arguments:

— The reason for the callback

— A pointer to the time of the callback

— The handle for the assertion

— A pointer to an attempt information structure

— A reference to the user data supplied when the callback was registered

The `t_vpi_attempt_info` attempt information structure contains details relevant to the specific event that occurred.

— On disable, enable, reset, and kill callbacks, the returned `p_vpi_attempt_info` info pointer is `NULL`, and no attempt information is available.

— On start and success callbacks, only the `attemptStartTime` field is valid.

— On a `cbAssertionFailure` callback, the `attemptStartTime` and `detail.failExpr` fields are valid.

— On a step callback, the `attemptStartTime` and `detail.step` fields are valid.

On a step callback, the `detail` describes the set of expressions matched in satisfying a step along the assertion, along with the corresponding source references. In addition, the `step` also identifies the source and destination "states" needed to uniquely identify the path being taken through the assertion. *State ids* are just integers, with `0` identifying the origin state, `1` identifying an accepting state, and any other number representing some intermediate point in the assertion. It is possible for the number of expressions in a step to be `0`, which represents an unconditional transition. In the case of a failing transition, the information provided is just as that for a successful one, but the last expression in the array represents the expression where the transition failed.

Details:

a)   In a failing transition, there shall always be at least one element in the expression array.

b)   Placing a step callback results in the same callback function being invoked for both success and failure steps.

c)   The content of the `cb_time` field depends on the reason identified by the reason field, as follows:

— `cbAssertionStart`: `cb_time` is the time when the assertion attempt has been started.

— `cbAssertionSuccess`, `cbAssertionFailure`: `cb_time` is the time when the assertion succeeded or failed.

— `cbAssertionStepSuccess`, `cbAssertionStepFailure`: `cb_time` is the time when the assertion attempt step succeeded or failed.

— `cbAssertionDisable, cbAssertionEnable, cbAssertionReset, cbAssertionKill`: `cb_time` is the time when the assertion attempt was disabled, enabled, reset, or killed.

d) In contrast to `cb_time`, the content of `attemptStartTime` is always the start time of the actual attempt of an assertion. It can be used as a unique identifier that distinguishes the attempts of any given assertion.

28.4 Control functions

This subclause defines how to obtain assertion system control and assertion control information.

28.4.1 Assertion system control

To control the assertion system, use `vpi_control()` with one of the following constants and a second handle argument that is either a `vpiHandle` for a scope or a `vpiCollection` of handles for a list of scopes. A `NULL` handle signifies that the control applies to all assertions regardless of scope.

— Usage example: `vpi_control(vpiAssertionSysReset, handle)`

`vpiAssertionSysReset` discards all attempts in progress for all assertions and restores the entire assertion system to its initial state. Any pre-existing `vpiAssertionStepSuccess` and `vpiAssertionStepFailure` callbacks shall be removed; all other assertion callbacks shall remain.

— Usage example: `vpi_control(vpiAssertionSysOff, handle)`

— `vpiAssertionSysOff` disables any further assertions from being started. Assertions already executing are not affected. This control has no effect on pre-existing assertion callbacks.

— `vpiAssertionSysKill` considers all attempts in progress as unterminated and disables any further assertions from being started. This control has no effect on pre-existing assertion callbacks.

— Usage example: `vpi_control(vpiAssertionSysOn, handle)`

`vpiAssertionSysOn` restarts the assertion system after it was stopped or suspended (e.g., due to `vpiAssertionSysOff` or `vpiAssertionSysKill`). Once started, attempts shall resume on all assertions. This control has no effect on prior assertion callbacks.

— Usage example: `vpi_control(vpiAssertionSysEnd, handle)`

`vpiAssertionSysEnd` discards all attempts in progress and disables any further assertions from starting. All assertion callbacks currently installed shall be removed. Once this control is issued, no further assertion-related actions shall be permitted.

28.4.2 Assertion control

To obtain assertion control information, use `vpi_control()` with one of the operators in this subclause.

For the following operators, the second argument shall be a valid assertion handle:

— Usage example: `vpi_control(vpiAssertionReset, assertionHandle)`

`vpiAssertionReset` discards all current attempts in progress for this assertion and resets this assertion to its initial state.

— Usage example: `vpi_control(vpiAssertionDisable, assertionHandle)`

`vpiAssertionDisable` disables the starting of any new attempts for this assertion. This has no effect on any existing attempts or if the assertion is already disabled. By default, all assertions are enabled.

— Usage example: `vpi_control(vpiAssertionEnable, assertionHandle)`

`vpiAssertionEnable` enables starting new attempts for this assertion. This has no effect on any existing attempts or if the assertion is already enabled.

For the following operators, the second argument shall be a valid assertion handle, and the third argument shall be an attempt start time (as a pointer to a correctly initialized `s_vpi_time` structure):

— Usage example: `vpi_control(vpiAssertionKill, assertionHandle, attemptStart-Time)`

`vpiAssertionKill` discards the given attempts, but leaves the assertion enabled and does not reset any state used by this assertion (e.g., `past()` sampling).

— Usage example: `vpi_control(vpiAssertionDisableStep, assertionHandle, attempt-StartTime)`

`vpiAssertionDisableStep` disables step callbacks for this assertion. This has no effect if stepping is not enabled or it is already disabled.

For the following operator, the second argument shall be a valid assertion handle, the third argument shall be an attempt start time (as a pointer to a correctly initialized `s_vpi_time` structure), and the fourth argument shall be a step control constant:

— Usage example: `vpi_control(vpiAssertionEnableStep, assertionHandle, attempt-StartTime, vpiAssertionClockSteps)`

  — `vpiAssertionEnableStep` enables step callbacks to occur for this assertion attempt. By default, stepping is disabled for all assertions. This call has no effect if stepping is already enabled for this assertion and attempt, other than possibly changing the stepping mode for the attempt if the attempt has not occurred yet. The stepping mode of any particular attempt cannot be modified after the assertion attempt in question has started.

  — The fine-grained step control constant `vpiAssertionClockSteps` indicates callbacks on a per-assertion/clock-tick basis. The assertion clock is the event expression supplied as the clocking expression to the assertion declaration. This step callback shall occur at every clocking event, when stepping is enabled, as the assertion "advances" in evaluation.

29. SystemVerilog code coverage control and API

29.1 Requirements

This clause defines the coverage API in SystemVerilog.

29.1.1 SystemVerilog API

The following criteria are used within this API:

a) This API shall be similar for all coverages. A wide number of coverage types are available, with possibly different sets offered by different vendors. Maintaining a common interface across all the different types enhances portability and ease of use.

b) At a minimum, the following types of coverage shall be supported:

1) Statement coverage

2) Toggle coverage

3) Finite state machine (FSM) coverage

    i) FSM states

    ii) FSM transitions

3) Assertion coverage

c) Coverage APIs shall be extensible in a transparent manner, i.e., adding a new coverage type shall not break any existing coverage usage.

d) This API shall provide means to obtain coverage information from specific subhierarchies of the design without requiring the user to enumerate all instances in those hierarchies.

29.1.2 Nomenclature

The following terms are used in this standard:

— **statement coverage:** Whether a statement has been executed. *Statement* is anything defined as a statement in the LRM. *Covered* means it executed at least once. Some implementations also permit querying the execution count. The granularity of statement coverage can be per-statement or per-statement block depending on the query (see 29.4.3 for details).

— **finite state machine (FSM) coverage:** The number of states in an FSM that this simulation reached. This standard does not require FSM automatic extraction, but a standard mechanism to force specific extraction is available via pragmas.

— **toggle coverage:** For each bit of every signal (wire and register), whether that bit has both a `0` value and a `1` value. *Full coverage* means both are seen; otherwise, some implementations can query for *partial coverage*. Some implementations also permit querying the toggle count of each bit.

— **assertion coverage:** For each assertion, whether it has had at least one success. Implementations permit querying for further details, such as attempt counts, success counts, failure counts and failure coverage.

These terms define the primitives for each coverage type. Over instances or blocks, the coverage number is merely the sum of all contained primitives in that instance or block.

29.2 SystemVerilog real-time coverage access

This subclause describes the mechanisms in SystemVerilog through which SystemVerilog code can query and control coverage information. Coverage information is provided to SystemVerilog by means of a

number of built-in system functions (described in <u>29.2.2</u>) using a number of predefined constants (described in <u>29.2.1</u>) to describe the types of coverage and the control actions to be performed.

29.2.1 Predefined coverage constants in SystemVerilog

The following predefined `defines represent basic real-time coverage capabilities accessible directly from SystemVerilog:

— Coverage control

```
`define SV_COV_START       0
`define SV_COV_STOP        1
`define SV_COV_RESET       2
`define SV_COV_CHECK       3
```

— Scope definition (hierarchy traversal/accumulation type)

```
`define SV_COV_MODULE      10
`define SV_COV_HIER        11
```

— Coverage type identification

```
`define SV_COV_ASSERTION   20
`define SV_COV_FSM_STATE   21
`define SV_COV_STATEMENT   22
`define SV_COV_TOGGLE      23
```

— Status results

```
`define SV_COV_OVERFLOW    -2
`define SV_COV_ERROR       -1
`define SV_COV_NOCOV       0
`define SV_COV_OK          1
`define SV_COV_PARTIAL     2
```

29.2.2 Built-in coverage access system functions

29.2.2.1 $coverage_control

```
$coverage_control(control_constant,
                  coverage_type,
                  scope_def,
                  modules_or_instance)
```

This function is used to control or query coverage availability in the specified portion of the hierarchy. The following control options are available:

— `SV_COV_START, if possible, starts collecting coverage information in the specified hierarchy. There is no effect if coverage is already being collected. Coverage is automatically started at the beginning of simulation for all portions of the hierarchy enabled for coverage.

— `SV_COV_STOP stops collecting coverage information in the specified hierarchy. There is no effect if coverage is not being collected.

— `SV_COV_RESET resets all available coverage information in the specified hierarchy. There is no effect if coverage not available.

— `SV_COV_CHECK` checks whether coverage information can be obtained from the specified hierarchy. The existence of coverage information does not imply that coverage is being collected, as the coverage could have been stopped.

The return value is a `define` name, with the value indicating the success of the action.

— `SV_COV_OK`, on a check operation, denotes that coverage is fully available in the specified hierarchy. For all other operations, it represents successful and complete execution of the desired operation.

— `SV_COV_ERROR`, on all operations, means that the control operation failed without any effect, typically due to errors in arguments, such as a nonexisting module.

— `SV_COV_NOCOV`, on a check or start operation, denotes that coverage is not available at any point in the specified hierarchy.

— `SV_COV_PARTIAL`, on a check or start operation, denotes that coverage is only partially available in the specified hierarchy.

Table 29-1 describes the possible return values for each of the coverage control options.

Table 29-1—Coverage control return values

	`SV_COV_OK`	`SV_COV_ERROR`	`SV_COV_NOCOV`	`SV_COV_PARTIAL`
`SV_COV_START`	Success	Bad args	No coverage	Partial coverage
`SV_COV_STOP`	Success	Bad args	—	—
`SV_COV_RESET`	Success	Bad args	—	—
`SV_COV_CHECK`	Full coverage	Bad args	No coverage	Partial coverage

Starting coverage on an instance that has already had coverage started via a prior call to `$coverage_control()` shall have no effect. Similarly, repeated calls to stop or reset coverage shall have no effect.

The hierarchy(ies) being controlled or queried are specified as follows:

— `SV_MODULE_COV, "unique module def name"` provides coverage of all instances of the given module (the unique module name is a string), excluding any child instances in the instances of the given module. The module definition name can use special notation to describe nested module definitions.

— `SV_COV_HIER, "module name"` provides coverage of all instances of the given module, including all the hierarchy below.

— `SV_MODULE_COV, instance_name` provides coverage of the one named instance. The instance is specified as a normal Verilog hierarchical path.

— `SV_COV_HIER, instance_name` provides coverage of the named instance, plus all the hierarchy below it.

All the permutations are summarized in Table 29-2.

Table 29-2—Instance coverage permutations

Control/query	Definition name	instance.name
`` `SV_COV_MODULE ``	The sum of coverage for all instances of the named module, excluding any hierarchy below those instances.	Coverage for just the named instance, excluding any hierarchy in instances below that instance.
`` `SV_COV_HIER ``	The sum of coverage for all instances of the named module, including all coverage for all hierarchy below those instances.	Coverage for the named instance and any hierarchy below it.
NOTE—Definition names are represented as strings, whereas instance names are referenced by hierarchical paths. A hierarchical path need not include any `.` if the path refers to an instance in the current context (i.e., normal Verilog hierarchical path rules apply).		

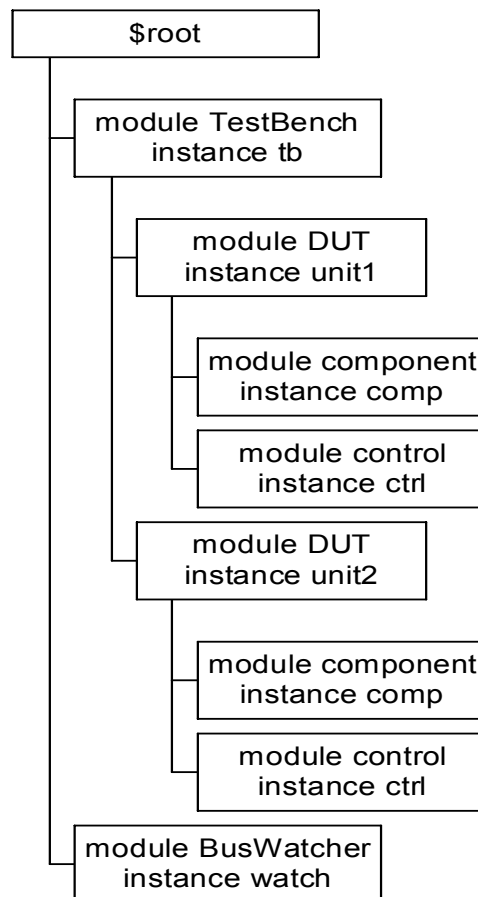See Figure 29-1 for an example of hierarchical instances.

Figure 29-1—Hierarchical instance example

If coverage is enabled on all instances shown in Figure 29-1, then

— `$coverage_control(`SV_COV_CHECK, `SV_COV_TOGGLE, `SV_COV_HIER, $root)` checks all instances to verify they have coverage and, in this case, returns `` `SV_COV_OK ``.

— `$coverage_control(`SV_COV_RESET, `SV_COV_TOGGLE, `SV_COV_MODULE, "DUT")` resets coverage collection on both instances of the DUT, specifically, `$root.tb.unit1` and `$root.tb.unit2`, but leaves coverage unaffected in all other instances.

— `$coverage_control(`SV_COV_RESET, `SV_COV_TOGGLE, `SV_COV_MODULE, $root.tb.unit1)` resets coverage of only the instance `$root.tb.unit1`, leaving all other instances unaffected.

— `$coverage_control(`SV_COV_STOP, `SV_COV_TOGGLE, `SV_COV_HIER, $root.tb.unit1)` resets coverage of the instance `$root.tb.unit1` and also resets coverage for all instances below it, specifically `$root.tb.unit1.comp` and `$root.tb.unit1.ctrl`.

— `$coverage_control(`SV_COV_START, `SV_COV_TOGGLE, `SV_COV_HIER, "DUT")` starts coverage on all instances of the module DUT and of all hierarchy(ies) below those instances. In this design, coverage is started for the instances `$root.tb.unit1`, `$root.tb.unit1.comp`, `$root.tb.unit1.ctrl`, `$root.tb.unit2`, `$root.tb.unit2.comp`, and `$root.tb.unit2.ctrl`.

29.2.2.2 $coverage_get_max

```
$coverage_get_max(coverage_type, scope_def, modules_or_instance)
```

This function obtains the value representing 100% coverage for the specified coverage type over the specified portion of the hierarchy. This value shall remain constant across the duration of the simulation.

NOTE—This value is proportional to the design size and structure; therefore, it also needs to be constant through multiple independent simulations and compilations of the same design, assuming any compilation options do not modify the coverage support or design structure.

The return value is an integer, with the following meanings:

— `-2 (`SV_COV_OVERFLOW)`. The value exceeds a number that can be represented as an integer.
— `-1 (`SV_COV_ERROR)`. An error occurred (typically due to using incorrect arguments).
— `0 (`SV_COV_NOCOV)`. No coverage is available for that coverage type on that/those hierarchy(ies).
— `+pos_num`. This value is the maximum coverage number (where $pos\_num > 0$), which is the sum of all coverable items of that type over the given hierarchy(ies).

The scope of this function is specified per `$coverage_control` (see 29.2.2.1).

29.2.2.3 $coverage_get

```
$coverage_get(coverage_type, scope_def, modules_or_instance)
```

This function obtains the current coverage value for the given coverage type over the given portion of the hierarchy. This number can be converted to a coverage percentage by use of the following equation:

$$coverage\% = \frac{coverage\_get()}{coverage\_get\_max()} *100$$

The return value follows the same pattern as `$coverage_get_max` (see 29.2.2.2), but with pos_num representing the current coverage level, i.e., the number of the coverable items that have been covered in this/these hierarchy(ies).

The scope of this function is specified per `$coverage_control` (see 29.2.2.1).

The return value is an integer, with the following meanings:

— `-2 (`SV_COV_OVERFLOW)`. The value exceeds a number that can be represented as an integer.

— -1 (`'SV_COV_ERROR`). An error occurred (typically due to using incorrect arguments).
— 0 (`'SV_COV_NOCOV`). No coverage is available for that coverage type on that/those hierarchy(ies).
— +*pos_num*. This value is the maximum coverage number (where *pos_num* > 0), which is the sum of all coverable items of that type over the given hierarchy(ies).

29.2.2.4 $coverage_merge

```
$coverage_merge(coverage_type, "name")
```

This function loads and merges coverage data for the specified coverage into the simulator. `name` is an arbitrary string used by the tool, in an implementation-specific way, to locate the appropriate coverage database, i.e., tools are allowed to store coverage files any place they want with any extension they want as long as the user can retrieve the information by asking for a specific saved name from that coverage database. If `name` does not exist or does not correspond to a coverage database from the same design, an error shall occur. If an error occurs during loading, the coverage numbers generated by this simulation might not be meaningful.

The return values from this function are as follows:
— `'SV_COV_OK`. The coverage data have been found and merged.
— `'SV_COV_NOCOV`. The coverage data have been found, but did not contain the coverage type requested.
— `'SV_COV_ERROR`. The coverage data were not found, or they did not correspond to this design, or another error occurred.

29.2.2.5 $coverage_save

```
$coverage_save(coverage_type, "name")
```

This function saves the current state of coverage to the tool's coverage database and associates it with the given name. This name will be mapped in an implementation-specific way into some file or set of files in the coverage database. Data saved to the database shall be retrieved later by using `$coverage_merge` and supplying the same name. Saving coverage shall not have any effect on the state of coverage in this simulation.

The return values from this function are as follows:
— `'SV_COV_OK`. The coverage data were successfully saved.
— `'SV_COV_NOCOV`. No such coverage is available in this design (nothing was saved).
— `'SV_COV_ERROR`. Some error occurred during the save. If an error occurs, the tool shall automatically remove the coverage database entry for `name` to preserve the coverage database integrity. It is not an error to overwrite a previously existing `name`.

Details:
a) The coverage database format is implementation dependent.
b) Mapping of names to actual directories or files is implementation dependent. There is no requirement that a coverage name map to any specific set of files or directories.

29.3 FSM recognition

Coverage tools need to have automatic recognition of many of the common FSM coding idioms in Verilog and SystemVerilog. This standard does not attempt to describe or require any specific automatic FSM recognition mechanisms. However, this standard does prescribe a means by which nonautomatic FSM extraction occurs. The presence of any of these standard FSM description additions shall override the tool's default extraction mechanism.

Identification of an FSM consists of identifying the following items:

— The state register (or expression)
— The next state register (this is optional)
— The legal states

29.3.1 Specifying signal that holds current state

Use the following pragma to identify the vector signal that holds the current state of the FSM:

```
/* tool state_vector signal_name */
```

where `tool` and `state_vector` are required keywords. This pragma needs to be specified inside the module definition where the signal is declared.

Another pragma is also required that specifies an enumeration name for the FSM. This enumeration name is also specified for the next state and any possible states, associating them with each other as part of the same FSM. There are two ways to do this:

— Use the same pragma as above:

```
/* tool state_vector signal_name enum enumeration_name */
```

— Use a separate pragma in the signal's declaration:

```
/* tool state_vector signal_name */
reg [7:0] /* tool enum enumeration_name */ signal_name;
```

In either case, **enum** is a required keyword. If using a separate pragma, `tool` is also a required keyword, and the pragma needs to be specified immediately after the bit range of the signal.

29.3.2 Specifying part-select that holds current state

A part-select of a vector signal can be used to hold the current state of the FSM. When a coverage tool displays or reports FSM coverage data, it names the FSM after the signal that holds the current state. If a part-select holds the current state in the user's FSM, the user needs to also specify a name for the FSM for the coverage tool to use. The FSM name is not the same as the enumeration name.

Specify the part-select by using the following pragma:

```
/* tool state_vector signal_name[n:n] FSM_name enum enumeration_name */
```

29.3.3 Specifying concatenation that holds current state

Like specifying a part-select, a concatenation of signals can be specified to hold the current state (when including an FSM name and an enumeration name):

```
/* tool state_vector {signal_name , signal_name, ...} FSM_name enum
   enumeration_name */
```

The concatenation is composed of all the signals specified. Bit-selects or part-selects of signals cannot be used in the concatenation.

29.3.4 Specifying signal that holds next state

The signal that holds the next state of the FSM can also be specified with the pragma that specifies the enumeration name:

```
reg [7:0] /* tool enum enumeration_name */
signal_name
```

This pragma can be omitted if, and only if, the FSM does not have a signal for the next state.

29.3.5 Specifying current and next state signals in same declaration

The tool assumes the first signal following the pragma holds the current state and the next signal holds the next state when a pragma is used for specifying the enumeration name in a declaration of multiple signals. For example:

```
/* tool state_vector cs */
reg [1:0] /* tool enum myFSM */ cs, ns, nonstate;
```

In this example, the tool assumes signal `cs` holds the current state and signal `ns` holds the next state. It assumes nothing about signal `nonstate`.

29.3.6 Specifying possible states of FSM

The possible states of the FSM can also be specified with a pragma that includes the enumeration name:

```
parameter /* tool enum enumeration_name */
S0 = 0,
s1 = 1,
s2 = 2,
s3 = 3;
```

Put this pragma immediately after the keyword parameter, unless a bit width for the parameters is used, in which case, specify the pragma immediately after the bit width:

```
parameter [1:0] /* tool enum enumeration_name */
S0 = 0,
s1 = 1,
s2 = 2,
s3 = 3;
```

29.3.7 Pragmas in one-line comments

These pragmas work in both block comments, between the `/*` and `*/` character strings, and one-line comments, following the `//` character string. For example:

```
parameter [1:0] // tool enum enumeration_name
S0 = 0,
s1 = 1,
s2 = 2,
s3 = 3;
```

29.3.8 Example

See for an example of FSM specified with pragmas.

```
module m3;

reg[31:0] cs;
reg[31:0] /* tool enum MY_FSM */ ns;
reg[31:0] clk;
reg[31:0] rst;

// tool state_vector cs enum MY_FSM

parameter // tool enum MY_FSM
p1=10,
p2=11,
p3=12;

endmodule // m3
```

Signal `ns` holds the next state

Signal `cs` holds the current state

`p1`, `p2`, and `p3` are possible states of

Figure 29-2—FSM specified with pragmas

29.4 VPI coverage extensions

29.4.1 VPI entity/relation diagrams related to coverage

29.4.2 Extensions to VPI enumerations

— Coverage control

```
vpiCoverageStart
vpiCoverageStop
vpiCoverageReset
vpiCoverageCheck
vpiCoverageMerge
vpiCoverageSave
```

— VPI properties

  — Coverage type properties

```
vpiAssertCoverage
vpiFsmStateCoverage
vpiStatementCoverage
vpiToggleCoverage
```

  — Coverage status properties

```
vpiCovered
vpiCoverMax
vpiCoveredCount
```

  — Assertion-specific coverage status properties

```
vpiAssertAttemptCovered
vpiAssertSuccessCovered
vpiAssertFailureCovered
```

  — FSM-specific methods

```
vpiFsmStates
vpiFsmStateExpression
```

— FSM handle types (vpi types)

```
vpiFsm
vpiFsmHandle
```

29.4.3 Obtaining coverage information

To obtain coverage information, the `vpi_get()` function is extended with additional VPI properties that can be obtained from existing handles:

```
vpi_get(<coverageType>, instance_handle)
```

returns the number of covered items of the given coverage type in the given instance. Coverage type is one of the coverage type properties described in 29.4.2. For example, given coverage type `vpiStatement-Coverage`, this call would return the number of covered statements in the instance pointed by `instance_handle`.

```
vpi_get(vpiCovered, assertion_handle)
vpi_get(vpiCovered, statement_handle)
vpi_get(vpiCovered, signal_handle)
vpi_get(vpiCovered, fsm_handle)
vpi_get(vpiCovered, fsm_state_handle)
```

returns whether the item referenced by the handle has been covered. For handles that can contain multiple coverable entities, such as statement, FSM, and signal handles, the return value indicates how many of the entities have been covered.

— For assertion handle, the coverable entities are assertions.
— For statement handle, the entities are statements.
— For signal handle, the entities are individual signal bits.
— For FSM handle, the entities are FSM states.

For assertions, `vpiCovered` implies that the assertion has been attempted, has succeeded at least once, and has never failed. More detailed coverage information can be obtained for assertions by the following queries:

```
vpi_get(vpiAssertAttemptCovered, assertion_handle)
```

returns the number of times the assertion has been attempted.

```
vpi_get(vpiAssertSuccessCovered, assertion_handle)
```

returns the number of times the assertion has succeeded nonvacuously or, if the assertion handle corresponds to a cover sequence, the number of times the sequence has been matched. Refer to 17.11.2 and 17.13 for the definition of vacuity.

```
vpi_get(vpiAssertVacuousSuccessCovered, assertion_handle)
```

returns the number of times the assertion has succeeded vacuously. Refer to 17.11.2 and 17.13 for the definition of vacuity.

```
vpi_get(vpiAssertFailureCovered, assertion_handle)
```

returns the number of times the assertion has failed. For any assertion, the number of attempts that have not yet reached any conclusion (success or failure) can be derived from the formula:

```
in progress = attempts - (successes + vacuous success + failures)
```

The example below illustrates some of these queries:

```
module covtest;
    bit on = 1, off = 0;
    logic clk;

    initial begin
        clk = 0;
        forever begin
            #10;
            clk = ~clk;
        end
    end

    always @(false) begin
        anvr: assert(on ##1 on); // assertion will not be attempted
    end

    always @(posedge clk) begin
        aundf: assert (on ##[1:$] off); // assertion will not pass or fail
        afail: assert (on ##1 off);     // assertion will always fail on 2nd
tick
        apass: assert (on ##1 on);      // assertion will succeed on each
attempt
    end
endmodule
```

For this example, the assertions will have the coverage results shown in Table 29-3.

Table 29-3—Assertion coverage results

	vpiCovered	vpiAssertAttempt-Covered	vpiAssertSuccess-Covered	vpiAssertFailure-Covered
anvr	False	False	False	False
aundf	False	True	False	False
afail	False	True	False	True
apass	True	True	True	False

The number of times an item has been covered can be obtained by the vpiCoveredCount property:

```
vpi_get(vpiCoveredCount, assertion_handle)
vpi_get(vpiCoveredCount, statement_handle)
vpi_get(vpiCoveredCount, signal_handle)
vpi_get(vpiCoveredCount, fsm_handle)
vpi_get(vpiCoveredCount, fsm_state_handle)
```

returns the number of times each coverable entity referred by the handle has been covered. The handle coverage information is only easily interpretable when the handle points to a unique coverable item (such as an individual statement). When the handle points to an item containing multiple coverable entities (such as a handle to a block statement containing a number of statements), the result is the sum of coverage counts for each of the constituent entities.

```
vpi_get(vpiCoveredMax, assertion_handle)
vpi_get(vpiCoveredMax, statement_handle)
```

```
vpi_get(vpiCoveredMax, signal_handle)
vpi_get(vpiCoveredMax, fsm_handle)
vpi_get(vpiCoveredMax, fsm_state_handle)
```

returns the number of coverable entities pointed by the handle. The number returned shall always be `1` when applied to an assertion or FSM state handle.

```
vpi_iterate(vpiFsm, instance-handle)
```

returns an iterator to all FSMs in an instance.

```
vpi_handle(vpiFsmStateExpression, fsm-handle)
```

returns the handle to the signal or expression encoding the FSM state.

```
vpi_iterate(vpiFsmStates, fsm-handle)
```

returns an iterator to all states of an FSM.

```
vpi_get_value(fsm_state_handle, state-handle)
```

returns the value of an FSM state.

29.4.4 Controlling coverage

Control of the collection of coverage shall be through the `vpi_control()` routine:

```
vpi_control(<coverageControl>, <coverageType>, instance_handle)
vpi_control(<coverageControl>, <coverageType>, assertion_handle)
```

Statement, toggle, and FSM coverage are not individually controllable (i.e., they are controllable only at the instance level and not on a per-statement, signal, or FSM basis). The semantics and behavior are per the `$coverage_control` system function (see 29.2.2.1). *coverageControl* shall be `vpiCoverageStart`, `vpiCoverageStop`, `vpiCoverageReset`, or `vpiCoverageCheck`, as defined in 29.4.2. *coverageType* is any one of the VPI coverage type properties (see 29.4.2)

To save coverage for the current simulation use:

```
vpi_control(coverageSave, <coverageType>, name)
```

as defined in 29.4.2. The semantics and behavior are specified per the equivalent system function `$coverage_save` (see 29.2.2.5).

To merge coverage for the current simulation use:

```
vpi_control(vpiCoverageMerge, <coverageType>, name)
```

as defined in 29.4.2. The semantics and behavior are specified per the equivalent system function `$coverage_merge` (see 29.2.2.4).

30. SystemVerilog data read API

30.1 Introduction

NOTE—This clause extends the SystemVerilog VPI with read facilities so that the VPI acts as an API for data access and tool interaction irrespective of whether the data are in memory or a persistent form such as a database and also irrespective of the tool with which the user is interacting.

SystemVerilog is both a design and verification language. Consequently, its VPI has a wealth of design and verification data access mechanisms. This makes the VPI an ideal vehicle for tool integration in order to replace arcane, inefficient, and error-prone file-based data exchanges with a new mechanism for tool-to-tool and user-to-tool interface. Moreover, a single-access API eases the interoperability investments for vendors and users alike. Reducing interoperability barriers allows vendors to focus on tool implementation. Users, on the other hand, are able to create integrated design flows from a multitude of best-in-class offerings spanning the realms of design and verification such as simulators, debuggers, formal, coverage, or testbench tools.

30.2 Requirements

SystemVerilog adds several design and verification constructs including the following:

— C data types such as **int**, **struct**, **union**, and **enum**

— Advanced built-in data types such as **string**

— User-defined data types and corresponding methods

— Data types and facilities that enhance the creation and functionality of testbenches

The access API shall be implemented by all tools as a minimal set for a standard means for user-tool or tool-tool interaction that involves SystemVerilog object data querying (reading). In other words, there is no need for a simulator to be running for this API to be in effect; it is a set of API routines that can be used for any interaction, e.g., between a user and a waveform tool to read the data stored in its database. This usage flow is shown in Figure 30-1.
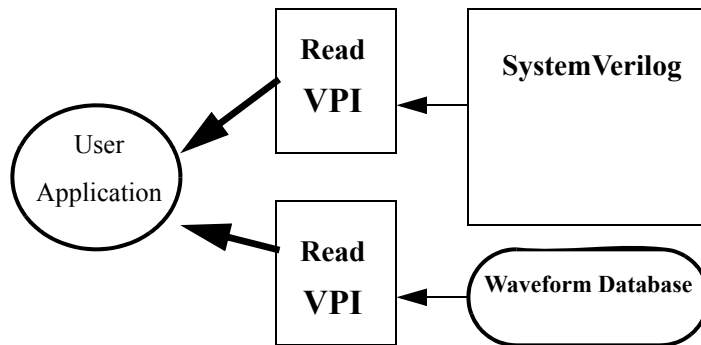

Figure 30-1—Data read VPI usage model

The focus in the API is the user view of access. While the API does provide varied facilities to give users the ability to effectively architect their applications, it does not address the tool-level efficiency concerns such as time-based incremental load of the data and/or predicting or learning the user access. It is left up to implementers to make this as easy and seamless as possible on the user. To make this easy on tools, the API provides an initialization routine where the user specifies access type and design scope. The user should be primarily concerned with the API specified here, and efficiency issues are dealt with behind the scenes.

30.3 Extensions to VPI enumerations

These extensions shall be appended to the contents of the `vpi_user.h` file, described in Annex G of IEEE Std 1364. The numbers in the range of 800 to 899 are reserved for the read data access portion of the VPI.

30.3.1 Object types

All objects in VPI have a `vpiType`. This API adds a new object type for data traversal (see 30.4.1) and three more objects types for object collection and traverse object collection (see 30.4.2). Examples for the usage of these object types can be found in 30.8.5, 30.8.6, and 30.8.7.

```
/* vpiHandle type for the data traverse object */
vpiTrvsObj                      /* Traverse object, use in vpi_handle() */
vpiCollection                   /* Collection of VPI handles */
vpiObjCollection                /* Collection of traversable design objs */
vpiTrvsCollection               /* Collection of vpiTrvsObj's */
```

The other object types that this API references, for example, to get a value at a specific time, are all the valid types in the VPI that can be used as arguments in the VPI routines for logic and strength value processing, such as `vpi_get_value(<object_handle>, <value_pointer>)`. These types include the following:

— Constants
— Nets and net arrays
— Variables
— Memories
— Parameters
— Primitives
— Assertions

In other words, any limitation in `vpiType` of `vpi_get_value()` shall also be reflected in this data access API.

30.3.2 Object properties

This subclause lists the object property VPI calls.

30.3.2.1 Static info

```
/* Check */
/* use in vpi_get() */
vpiIsLoaded                     /* is loaded */
vpiHasDataVC                    /* has at least one VC
                                        at some point in time
                                        in the database */
vpiHasVC                        /* has VC at specific time */
vpiHasNoValue                   /* has no value at specific time */
vpiBelong                       /* belong to extension */

/* Access */
vpiAccessLimitedInteractive     /* interactive */
vpiAccessInteractive            /* interactive: history */
vpiAccessPostProcess            /* database */
/* Iteration on instances for loaded */
vpiDataLoaded                   /* use in vpi_iterate() */
```

30.3.2.2 Dynamic info

30.3.2.2.1 Control constants

```
/* Control Traverse: use in vpi_goto() for a vpiTrvsObj type */
vpiMinTime                              /* min time */
vpiMaxTime                              /* max time */
vpiPrevVC                               /* previous value change */
vpiNextVC                               /* next value change */
vpiTime                                 /* time jump */
```

These properties can also be used in `vpi_get_time()` to enhance the access efficiency. The routine `vpi_get_time()` with a traverse handle argument is extended with the additional ability to get the minimum, maximum, previous value change, and next value change times of the traverse handle, not just the current time of the handle. These same control constants can then be used for access and for moving the traverse handle where the context (get or go to) can distinguish the intent.

30.3.3 System callbacks

The access API adds no new system callbacks. The reader routines (methods) can be called whenever the user application has control and wishes to access data.

30.4 VPI object type additions

30.4.1 Traverse object

To access the value changes of an object over time, the notion of a value change traverse handle is added. A value change traverse object is used to traverse and access value changes not just for the current value (as calling `vpi_get_time()` or `vpi_get_value()` on the object handle would) but also for any point in time: past, present, or future. To create a value change traverse handle, the routine `vpi_handle()` is called with a `vpiTrvsObj` `vpiType`:

```
vpiHandle object_handle; /* design object */
vpiHandle trvsHndl = vpi_handle(/*vpiType*/vpiTrvsObj,
                                /*vpiHandle*/ object_handle);
```

A traverse object exists from the time it is created until its handle is released. It is the application's responsibility to keep a handle to the created traverse object and to release it when it is no longer needed.

30.4.2 VPI collection

In order to read data efficiently, it can be necessary to specify a group of objects. For example, when traversing data, a user might wish to specify a list of objects to be marked as targets of data traversal. To do this grouping requires the notion of a collection. A collection represents a user-defined collection of VPI handles. The collection is an ordered list of VPI handles. The `vpiType` of a collection handle can be `vpiCollection`, `vpiObjCollection`, or `vpiTrvsCollection`:

— A collection of type `vpiCollection` is a general collection of VPI handles of objects of any type.

— The collection object of type `vpiObjCollection` represents a collection of VPI traversable objects in the design.

— A `vpiTrvsCollection` is a collection of traverse objects of type `vpiTrvsObj`.

The usage here in the read API is either one of the following:

— Collections of traversable design objects: Used, for example, in `vpi_handle()` to create traverse handles for the collection. A collection of traversable design objects is of type `vpiObjCollection` (the elements can be any object type in the design except traverse objects of type `vpiTrvsObj`).

— Collections of data traverse objects: Used, for example, in `vpi_goto()` to move the traverse handles of all the objects in the collection (all are of type `vpiTrvsObj`). A collection of traverse objects is a `vpiTrvsCollection`.

The collection contains a set of member VPI objects and can take on an arbitrary size. The collection can be created at any time, and existing objects can be added to it. The reader implementation can perform a type check on the items being added to the collection and generate an error if the item added does not belong to the allowed `vpiType`.

The purpose of a collection is to group design objects and permit operating on each element with a single operation applied to the whole collection group. `vpi_iterate(vpiMember, <collection_handle>)` is used to create a member iterator. `vpi_scan()` can then be used to scan the iterator for the elements of the collection.

A collection object is created with `vpi_create()`. The first call provides `NULL` handles to the collection object and the object to be added. Following calls, which can be performed at any time, provide the collection handle and a handle to the object for addition. The return argument is a handle to the collection object.

For example:

```
vpiHandle designCollection;
vpiHandle designObj;
vpiHandle trvsCollection;
vpiHandle trvsObj;
/* Create a collection of design objects    */
designCollection = vpi_create(vpiObjCollection, NULL, NULL);
/* Add design object designObj into it      */
designCollection = vpi_create(vpiObjCollection, designCollection, designObj);

/* Create a collection of traverse objects*/
trvsCollection = vpi_create(vpiTrvsCollection, NULL, NULL);
/* Add traverse object trvsObj into it        */
trvsCollection = vpi_create(vpiTrvsCollection, trvsCollection, trvsObj);
```

Sometimes it is necessary to filter a collection and extract a set of handles that meet, or do not meet, a specific criterion for a given collection. The function `vpi_filter()` can be used for this purpose in the following form:

```
vpiHandle colFilterHdl = vpi_filter((vpiHandle) colHdl, (PLI_INT32)
    filterType, (PLI_INT32) flag);
```

The first argument of `vpi_filter()`, `colHdl`, shall be the collection on which to apply the filter operation. The second argument, `filterType`, can be any `vpiType` or VPI boolean property. This argument is the criterion used for filtering the collection members. The third argument, `flag`, is a boolean value. If set to `TRUE`, `vpi_filter()` shall return a collection of handles that match the criterion indicated by `filterType`. If set to `FALSE`, `vpi_filter()` shall return a collection of handles that do not match the criterion indicated by `filterType`. The original collection passed as a first argument remains unchanged.

A collection object exists from the time it is created until its handle is released. It is the application's responsibility to keep a handle to the created collection and to release it when it is no longer needed.

30.4.2.1 Operations on collections

A traverse collection can be obtained (i.e., created) from a design collection using `vpi_handle()`. The call would take on the following form:

```
vpiHandle objCollection;
/* Obtain a traverse collection from the object collection */
vpi_handle(vpiTrvsCollection, objCollection);
```

The usage of this capability is discussed in 30.8.7.

Another optional method is defined, which is used when the user wishes to directly control the data load, for loading data of objects in a collection: `vpi_load()`. This operation loads all the objects in the collection. It is equivalent to performing a `vpi_load()` on every single handle of the object elements in the collection.

A traversal method is also defined on collections of traverse handles, i.e., collections of type `vpiTrvsCollection`. The method is `vpi_goto()`.

30.5 Object model diagrams

A traverse object of type `vpiTrvsObj` is related to its parent object; it is a means to access the value data of said object. An object can have several traverse objects, each pointing and moving in a different way along the value data horizon. This is shown graphically in Figure 30-2. The traversable class is a representational grouping consisting of any object that

— Has a name.

— Can take on a value accessible with `vpi_get_value()`. The value must be variable over time (i.e., necessitates creation of a traverse object to access the value over time).

The class includes nets, array nets, variables, memories, primitives, primitive arrays, assertions, and parameters. It also includes part-selects of all the design object types that can have part-selects.

A collection object of type `vpiObjCollection` groups together a set of design objects `Obj` (of any type). A traverse collection object of type `vpiTrvsCollection` groups together a set of traverse objects `trvsObj` of type `vpiTrvsObj`. See Figure 30-3.

30.6 Usage extensions to VPI routines

Several VPI routines, which existed before SystemVerilog, have been extended in usage with the addition of new object types and/or properties. While the extensions are fairly obvious, they are emphasized in Table 30-1 again to turn the reader's attention to the extended usage.
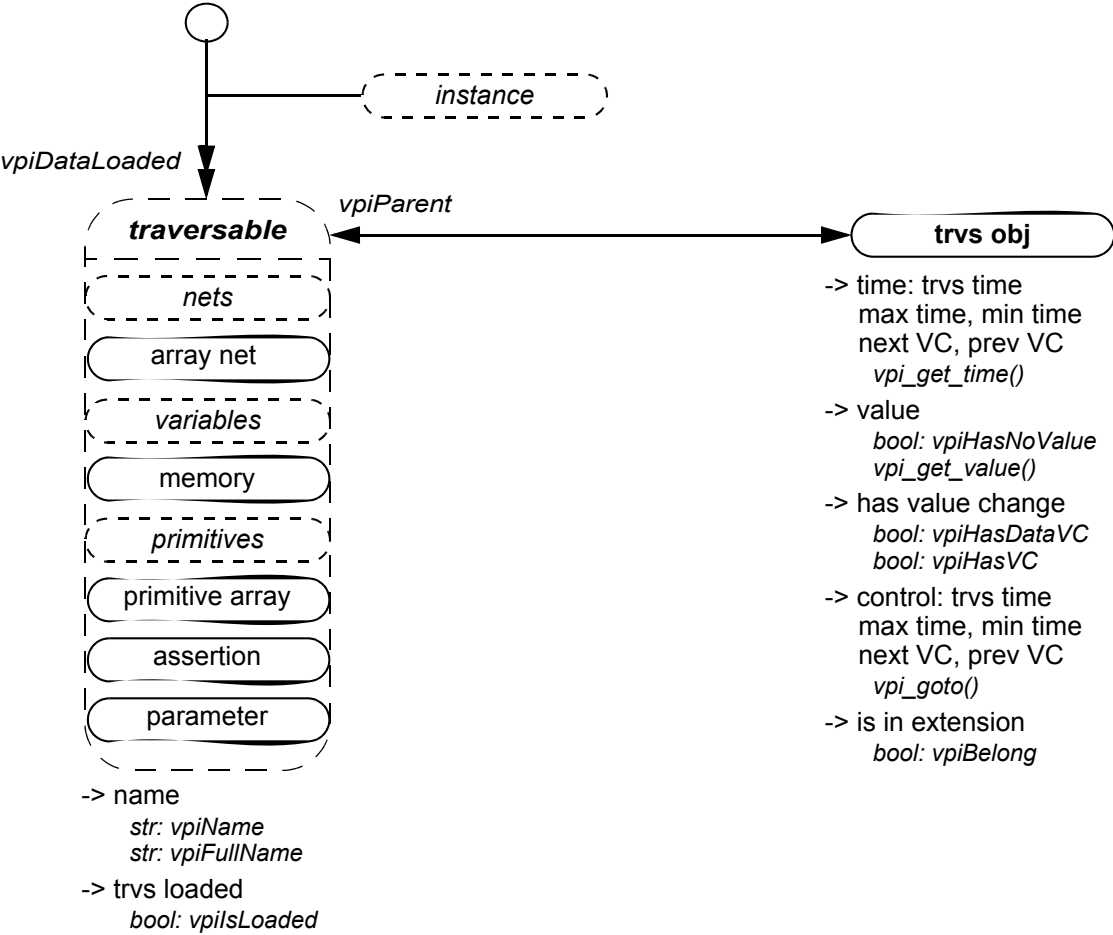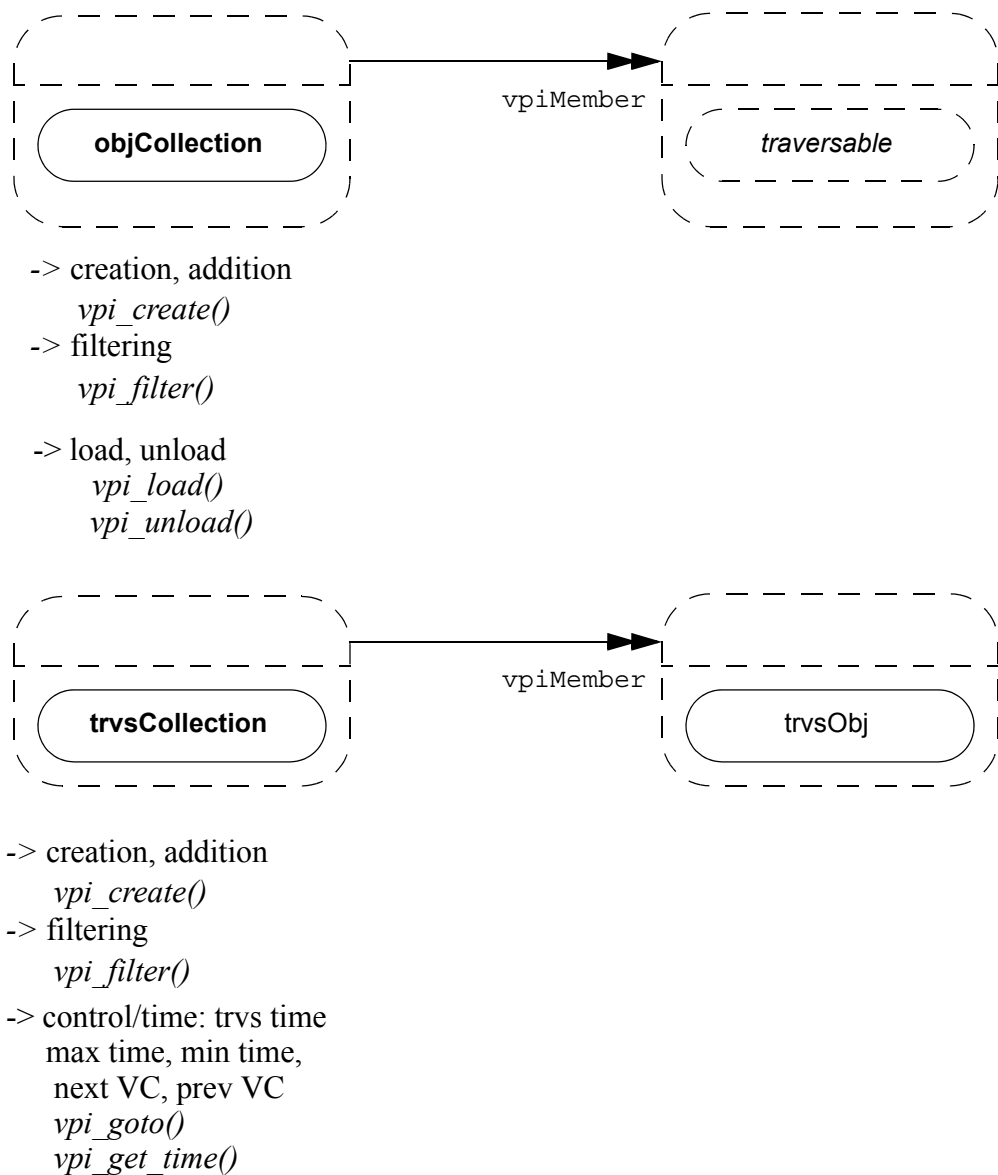
Figure 30-2—Model diagram of traverse object

Figure 30-3—Model diagram of collection

Table 30-1—Usage extensions to Verilog VPI routines

To	Use	New usage
Get tool's reader version	`vpi_get_vlog_info()`	Reader version information
Create an iterator for the loaded objects (using `vpi_iterate(vpiData-Loaded, <instance>)`). Create an iterator for (object or traverse) collections using `vpi_iterate(vpiMember, <collection>)`.	`vpi_iterate()`	Add iteration types `vpiData-Loaded` and `vpiMember`. Extended with collection handle to create a collection member element iterator.

Table 30-1—Usage extensions to Verilog VPI routines *(continued)*

To	Use	New usage
Obtain a traverse (collection) handle from an object (collection) handle	`vpi_handle()`	Add new types `vpiTrvsObj` and `vpiTrvsCollection`. Extended with collection handle (of traversable objects) to create a traverse collection from an object collection.
Obtain a property.	`vpi_get()`	Extended with the new check properties: `vpiIsLoaded`, `vpiHasDataVC`, `vpiHasVC`, `vpiHasNoValue`, and `vpiBelong`.
Get a value.	`vpi_get_value()`	Use traverse handle as argument to get value where handle points.
Get time traverse (collection) handle points at.	`vpi_get_time()`	Use traverse (collection) handle as argument to get current time where handle points. Also, get the traverse handle minimum time, maximum time, previous value change time, or next value change time.
Free traverse handle. Free (traverse) collection handle.	`vpi_free_object()`	Use traverse handle as argument. Use (traverse) collection handle as argument.

30.7 VPI routines added in SystemVerilog

Table 30-2 and Table 30-3 list all the VPI routines added in SystemVerilog.

Table 30-2—VPI routines

To	Use
For the reader extension, initialize read interface by loading the appropriate reader extension library (simulator, waveform, or other tool). All VPI routines defined by the reader extension library shall be called by indirection through the returned pointer; only built-in VPI routines can be called directly.	`vpi_load_extension()`

Table 30-3—Reader VPI routines

To	Use
Perform any tool cleanup. Close database (if opened in `vpiAccessPostProcess` or `vpiAccessInteractive` mode).	`vpi_close()`
Create a new handle: used to — Create an object (traverse) collection. — Add a (traverse) object to an existing collection.	`vpi_create()`

Table 30-3—Reader VPI routines *(continued)*

To	Use
Filter a collection and extract a set of handles that meet, or do not meet, a specific criterion for a given collection.	`vpi_filter()`
Move traverse (collection) to minimum, maximum, or specific time. Return a new traverse (collection) handle containing all the objects that have a value change at that time.	`vpi_goto()`
Load data (for a single design object or a collection) onto memory if the user wishes to exercise this level of data load control.	`vpi_load()`
Initialize load access.	`vpi_load_init()`
Unload data (for a single design object or a collection) from memory if the user wishes to exercise this level of data load control.	`vpi_unload()`

30.8 Reading data

Reading data is performed in three steps:

a) A design object must be selected for traverse access from a database (or from memory).

b) The intent to access data must be indicated. This is typically done by a `vpi_load_init()` call as a hint from the user to the tool on which areas of the design are going to be accessed. The tool shall then load the data in an invisible fashion to the user (for example, right after the call, or at traverse handle creation, or at usage). Alternatively, `vpi_load()` may be called (this can be done at any point in time) to load, or force the load of, a specified object or collection of objects. This can be done either instead of or in addition to the objects in the scope or collection specified in `vpi_load_init()`). `vpi_unload()` can be used by the user to force the tool to unload specific objects. It should be noted that traverse handle creation shall fail for unloaded objects or collections.

c) Once an object is selected and marked for load, a traverse object handle can be created and used to traverse the design objects' stored data.

At this point the object is available for reading. The traverse object permits the data value traversal and access.

30.8.1 VPI read initialization and load access initialization

Selecting an object is done in three steps:

a) The first step is to initialize the read access with a call to `vpi_load_extension()` to load the reader extension and set the following:

— Name of the reader library to be used specified as a character string. This is either a full pathname to this library or the single filename (without path information) of this library, assuming a vendor-specific way of defining the location of such a library. The latter method is more portable and, therefore, recommended. Neither the full pathname nor the single filename shall include an extension. The name of the library must be unique, and the appropriate extension for the actual platform should be provided by the application loading this library. More details are in 30.10.

— Name of the database holding the stored data or flush database in case of `vpiAccessPost-Process` or `vpiAccessInteractive`, respectively; a `NULL` can be used in case of `vpiAccessLimitedInteractive`. This is the logical name of a database, not the name of a file in the file system. It is implementation dependent whether there is any relationship to an

actual on-disk object and the provided name. See access mode below for more details on the access modes.

— Access mode: The following VPI properties set the mode of access:

— `vpiAccessLimitedInteractive`: The access shall be done for the data stored in the tool memory (e.g., simulator), and the history (or future) that the tool stores is implementation dependent. If the tool does not store the requested info, then the querying routines shall return a failure. The database name argument to `vpi_load_extension()` in this mode shall be ignored (even if not `NULL`).

— `vpiAccessInteractive`: The access shall be done interactively. The tool shall then use the database specified as a "flush" area for its data. This mode is very similar to the `vpiAccessLimitedInteractive` with the additional requirement that all the past history (before current time) shall be stored (for the specified scope or collection). See the access scope and collection description of `vpi_load_init()`.

— `vpiAccessPostProcess`: The access shall be done through the specified database. All data queries shall return the data stored in the specified database. Data history depends on what is stored in the database and can be nothing (i.e., no data).

`vpi_load_extension()` can be called multiple times for different reader interface libraries (coming from different tools), database specification, and/or read access. A call with `vpiAccessInteractive` means that the user is querying the data stored inside the simulator database and uses the VPI routines supported by the simulator. A call with `vpiAccessPostProcess` means that the user is accessing the data stored in the database and uses the VPI services provided by the waveform tool. The application, if accessing several databases and/or using multiple read API libraries, can use the routine `vpi_get(vpiBelong, <vpiHandle>)` to check whether a handle belongs to that database. The call is performed as follows:

```
reader_extension_ptr->vpi_get(vpiBelong, <vpiHandle>);
```

where `reader_extension_ptr` is the reader library pointer returned by the call to `vpi_load_extension()`. `TRUE` is returned if the passed handle belongs to that extension, and `FALSE` otherwise. If the application uses the built-in library (i.e., the one provided by the tool under which it is running), there is no need to use indirection to call the VPI routines; they can be called directly. An initial call must, however, be made to set the access mode, specify the database, and check for error indicated by a `NULL` return.

`vpi_close()` shall be called in the following cases:

— `vpiAcessLimitedInteractive` to perform any tool cleanup. The validity of VPI handles after this call is left up to the particular reader implementation.

— `vpiAccessPostProcess` or `vpiAccessInteractive` mode to perform any tool cleanup and close the opened database. Handles obtained before the call to `vpi_close()` are no longer valid after this call.

Multiple databases, possibly in different access modes (for example, a simulator database opened in `vpiAccessInteractive` and a database opened in `vpiAccessPostProcess`, or two different databases opened in `vpiAccessPostProcess`) can be accessed at the same time. In 30.10, an example is shown of how to access multiple databases from multiple read interfaces simultaneously.

b) Next step is to specify the elements that shall be accessed. This is accomplished by calling `vpi_load_init()` and specifying a scope and/or an item collection. At least one of the two (scope or collection) needs to be specified. If both are specified, then the union of all the object elements forms the entire set of objects the user can access.

— Access scope: The specified scope handle and nesting mode govern the scope that access returns. Data queries outside this scope (and its subscopes as governed by the nesting mode) shall return a fail in the access routines unless the object belongs to the access collection

described below. It can be used either in a complementary or in an exclusive fashion to the access collection. NULL is to be passed to the collection when the access scope is used in an exclusive fashion.

— Access collection: The specified collection stores the traverse object handles to be loaded. It can be used either in a complementary or in an exclusive fashion to the access scope. NULL is to be passed to the scope when the access collection is used in an exclusive fashion.

vpi_load_init() enables access to the objects stored in the database and can be called multiple times. The load access specification of a call remains valid until the next call is executed. This routine serves to initialize the tool load access and provides an entry point for the tool to perform data access optimizations.

30.8.2 Object selection for traverse access

In order to select an object for access, the user must first obtain the object handle. This can be done using the VPI routines (that are supported in the tool being used) for traversing the HDL hierarchy and obtaining an object handle based on the type of object relationship to another (top) handle.

Any tool that implements this read API (e.g., waveform tool) shall implement at least a basic subset of the design navigation VPI routines that shall include vpi_handle_by_name() to permit the user to get a vpiHandle from an object name. It is left up to tool implementation to support additional design navigation relationships. Therefore, if the application wishes to access similar elements from one database to another, it shall use the name of the object and then call vpi_handle_by_name() to get the object handle from the relevant database. This level of indirection is always safe to do when switching the database query context and shall be guaranteed to work.

It should be noted that an object's vpiHandle depends on the access mode specified in vpi_load_extension() and the database accessed (identified by the returned extension pointer; see 30.10). A handle obtained through a post-process access mode (vpiAccessPostProcess) from a waveform tool, for example, is not interchangeable in general with a handle obtained through interactive access mode (vpiAccessLimitedInteractive or vpiAccessInteractive) from a simulator. Also handles obtained through post-process access mode of different databases are not interchangeable. This is because objects, their data, and their relationships in a stored database could be quite different from those in the simulation model and those in other databases.

30.8.3 Optionally loading objects

As mentioned earlier, vpi_load_init() allows the tool implementing the reader to load objects in a fashion that is invisible to the user. Optionally, if users choose to do their own loading at some point in time, then once the object handle is obtained they can use the VPI data load routine vpi_load() with the object's vpiHandle to load the data for the specific object onto memory. Alternatively, for efficiency considerations, vpi_load() can be called with a design object collection handle of type vpiObjCollection. The collection must have already been created with vpi_create() and the (additional) selected object handles added to the load collection using vpi_create() with the created collection list passed as argument. The object data are not accessible as of yet to the user's read queries; a traverse handle must still be created. This is presented in 30.8.4.

The phrase *loading the object* means loading the object from a database into memory or marking it for active use if it is already in the memory hierarchy. Object loading is the portion that tool implementers need to look at for efficiency considerations. Reading the data of an object, if loaded in memory, is a simple consequence of the load initialization (vpi_load_init()) and/or vpi_load() optionally called by the user. The API does not specify here any memory hierarchy or caching strategy that governs the access (load or read) speed. It is left up to tool implementation to choose the appropriate scheme. It is recommended that this happens in a fashion invisible to the user without requiring additional routine calls.

The API here provides the tool with the chance to prepare itself for data load and access with the `vpi_load_init()`. With this call, the tool can examine what objects the user wishes to access before the actual read access is made. The API also provides the user with the ability to force loads and unloads, but it is recommended to leave this to the tool unless there is a need for the user application to influence this aspect.

30.8.3.1 Iterating design for loaded objects

The user shall be allowed to optionally iterate for the loaded objects in a specific instantiation scope using `vpi_iterate()`. This shall be accomplished by calling `vpi_iterate()` with the appropriate reference handle and using the property `vpiDataLoaded` as follows:

a) Iterate on all data read loaded objects in the design: use a `NULL` reference handle (`ref_h`) to `vpi_iterate()`. For example:

```
itr = vpi_iterate(vpiDataLoaded, /* ref_h */ NULL);
while (loadedObj = vpi_scan(itr)) {
   /* process loadedObj */
}
```

b) Iterate on all data read loaded objects in an instance: pass the appropriate instance handle as a reference handle to `vpi_iterate()`. For example:

```
itr = vpi_iterate(vpiDataLoaded, /* ref_h */ instanceHandle);
while (loadedObj = vpi_scan(itr)) {
   /* process loadedObj */
}
```

30.8.3.2 Iterating object collection for its member objects

The user shall be allowed to iterate for the design objects in a design collection using `vpi_iterate()` and `vpi_scan()`. This shall be accomplished by creating an iterator for the members of the collection and then using `vpi_scan()` on the iterator handle. For example:

```
vpiHandle var_handle;   /* some object          */
vpiHandle varCollection;/* object collection    */
vpiHandle Var;          /* object handle        */
vpiHandle itr;          /* iterator handle      */
/* Create object collection                     */
varCollection = vpi_create(vpiObjCollection, NULL, NULL);
/* Add elements to the object collection        */
varCollection = vpi_create(vpiObjCollection, varCollection, var_handle);

/* Iterating a collection for its elements */
itr = vpi_iterate(vpiMember, varCollection); /* create iterator   */
while (Var = vpi_scan(itr)) {                 /* scan iterator     */
/* process Var */
}
```

30.8.4 Reading an object

The subclauses above have outlined:

— How to select an object for access, in other words, marking this object as a target for access. This is where the design navigation VPI is used.

— How to call `vpi_load_init()` as a hint on the areas to be accessed and/or optionally to load an object into memory after obtaining a handle and then loading objects either individually or as a group using the object collection.

— How to optionally iterate the design scope and the object collection to find the loaded objects if needed.

In this subclause, reading data is discussed. Reading an object's data means obtaining its value changes. VPI, before this extension, had allowed a user to query a value at a specific point in time (namely, the current time), and its access does not require the extra step of giving a load hint or actually loading the object data. This step is added here because VPI is extended with a temporal access component: The user can ask about all the values in time (regardless of whether that value is available to a particular tool or found in memory or a database, the mechanism is provided) because accessing this value horizon involves a larger memory expense and possibly a considerable access time. This subclause describes how to access and traverse this value timeline of an object.

To access the value changes of an object over time, a traverse object is used, as introduced earlier in 30.4.1. Several VPI routines are also added to traverse the value changes (using this new handle) back and forth. This mechanism is very different from the "iteration" notion of VPI that returns objects related to a given object; the traversal here can walk or jump back and forth on the value change timeline of an object. To create a value change traverse handle, the routine `vpi_handle()` must be called in the following manner:

```
vpiHandle trvsHndl = vpi_handle(vpiTrvsObj, object_handle);
```

A user (or tool) application can create more than one value change traverse handle for the same object, thus providing different views of the value changes. Each value change traverse handle shall have a means to have an internal index, which is used to point to its "current" time and value change of the place to which it points. In fact, the value change traversal can be done by increasing or decreasing this internal index. What this index is and how its function is performed are left up to tool implementation; it is only used as a concept for explanation here.

Once created, the traverse handle can point anywhere along the timeline; its initial location is left for tool implementation. However, if the traverse object has no value changes, the handle shall point to the minimum time (of the trace) so that calls to `vpi_get_time()` can return a valid time. It is up to the user to call an initial `vpi_goto()` to move to the desired initial pointing location.

30.8.4.1 Traversing value changes of objects

After getting a traverse `vpiHandle`, the application can do a forward or backward walk or jump traversal by using `vpi_goto()` on a `vpiTrvsObj` object type with the new traverse properties.

Example of code segment for the complete process from handle creation to traversal:

```
p_vpi_extension reader_p;  /* Pointer to VPI reader extension structure */
vpiHandle instanceHandle;  /* Some scope object is inside                */
vpiHandle var_handle;      /* Object handle                             */
vpiHandle vc_trvs_hdl;     /* Traverse handle                           */
vpiHandle itr;
p_vpi_value value_p;       /* Value storage                             */
p_vpi_time time_p;         /* Time storage                              */
PLI_INT32 code;            /* return code                               */
...
/* Initialize the read interface: Access data from memory            */
/* NOTE: Use built-in VPI (e.g., that of simulator application is running
   under)                                                            */
reader_p = vpi_load_extension(NULL, NULL, vpiAccessLimitedInteractive);
```

```
   if (reader_p == NULL) ... ; /* Not successful */


   /* Initialize the load: Access data from simulator) memory, for scope
   instanceHandle and its subscopes */
   /* NOTE: Call marks access for all the objects in the scope */
   vpi_load_init(NULL, instanceHandle, 0);



   itr = vpi_iterate(vpiVariables, instanceHandle);
   while (var_handle = vpi_scan(itr)) {
   /* Demo how to force the load, this part can be skipped in general */
      if (vpi_get(vpiIsLoaded, var_handle) == 0) { /* not loaded*/
         /* Load data: object-based load, one by one */
         if (!vpi_load(var_handle)); /* Data not found !        */
            break;
      }
   /*-- End of Demo how to force the load, this part can be skipped in general */
      /* Create a traverse handle for read queries */
      vc_trvs_hdl = vpi_handle(vpiTrvsObj, var_handle);
      /* Go to minimum time */
      vc_trvs_hdl = vpi_goto(vpiMinTime, vc_trvs_hdl, NULL, NULL);
      /* Get info at the min time */
      time_p->type = vpiSimTime;
      vpi_get_time(vc_trvs_hdl, time_p); /* Minimum time */
      vpi_printf(...);
      vpi_get_value(vc_trvs_hdl, value_p); /* Value */
      vpi_printf(...);
      if (vpi_get(vpiHasDataVC, vc_trvs_hdl)) { /* Have any VCs ? */
         for (;;) { /* All the elements in time */
            vc_trvs_hdl = vpi_goto(vpiNextVC, vc_trvs_hdl, NULL, &code);
            if (!code) {
               /* failure (e.g., already at MaxTime or no more VCs) */
               break; /* cannot go further */
            }
            /* Get Max time: Set bits of s_vpi_time type field */
            /* time_p->type = vpiMaxTime & vpiSimTime; */
            /* vpi_get_time(vc_trvs_hdl, time_p); */
            time_p->type = vpiSimTime;
            vpi_get_time(vc_trvs_hdl, time_p);     /* Time of VC  */
            vpi_get_value(vc_trvs_hdl, value_p);  /* VC data     */
         }
      }
   }
   /* free handles */
   vpi_free_object(...);
```

The code segment above declares an interactive access scheme, where only a limited history of values is provided by the tool (e.g., simulator). It then creates a value change traverse handle associated with an object whose handle is represented by `var_handle`, but only after `vpi_load_init()` is called. It then creates a traverse handle, `vc_trvs_hdl`. With this traverse handle, it first calls `vpi_goto()` to move to the minimum time where the value has changed. It moves the handle (internal index) to that time by calling `vpi_goto()` with a `vpiMinTime` argument. It then repeatedly calls `vpi_goto()` with a `vpiNextVC` to move the internal index forward repeatedly until there is no value change left. `vpi_get_time()` gets the actual time where this value change is, and data are obtained by `vpi_get_value()`. The application can also choose to call `vpi_goto()` with a `time_p` argument to automatically get the value change time instead of calling `vpi_get_time()` separately to get this information.

The traverse and collection handles can be freed when they are no longer needed using `vpi_free_object()`.

30.8.4.2 Jump behavior

Jump behavior refers to the behavior of `vpi_goto()` with a `vpiTime` control constant, `vpiTrvsObj` type, and a jump time argument. The time specified is the time to which the traverse handle should jump. The specified time might not have value changes. In that case, the traverse handle shall point to the latest value change equal to or less than the time requested.

In the example below, the whole simulation run is from time 10 to time 65, and a variable has value changes at time 10, 15, and 50. If a value change traverse handle is associated with this variable and a jump to a different time is attempted, the result shall be determined as follows:

— Jump to 12; traverse handle return time is 10.
— Jump to 15; traverse handle return time is 15.
— Jump to 65; traverse handle return time is 50.
— Jump to 30; traverse handle return time is 15.
— Jump to 0; traverse handle return time is 10.
— Jump to 50; traverse handle return time is 50.

If the jump time has a value change, then the internal index of the traverse handle shall point to that time. Therefore, the return time is exactly the same as the jump time.

If the jump time does not have a value change and if the jump time is not less than the minimum time of the whole trace[8] run, then the return time is aligned backward. If the jump time is less than the minimum time, then the return time shall be the minimum time. When the object has hold value semantics between the value changes such as static variables, then the return of `vpi_goto()` (with a specified time argument to jump to) is a new handle pointing to that time to indicate success. When the time is greater than the trace maximum time or when an automatic object or an assertion or any other object does not hold its value between the value changes, then the return code should indicate failure (and the backward time alignment is still performed). In other words, the time returned by the traverse object shall never exceed the trace maximum; the maximum point in the trace is not marked as a value change unless there is truly a value change at that point in time (see the example in this subclause).

30.8.4.3 Dump-off regions

When accessing a database, it is likely that there are gaps along the value timeline where possibly the data recording (e.g., dumping from simulator) was turned off. In this case, the starting point of that interval shall be marked as a value change if the object had a stored value before that time. `vpi_goto()`, whether used to jump to that time or using next value change or previous value change traversal from a point before or after, respectively, shall stop at that value change. Calling `vpi_get_value()` on the traverse object pointing to that value change shall have no effect on the value argument passed; the time argument shall be filled with the time at that value change. `vpi_get()` can be called in the form `vpi_get(vpiHasNoValue, <traverse handle>)` to return `TRUE` if the traverse handle has no value (i.e., pointing to the start of a dump off region) and `FALSE` otherwise.

There is, of course, another value change (from no recorded value to an actual recorded value) at the end of the dump-off interval, if the end exists, i.e., there is additional dumping performed and data for this object

---

[8]The word trace can be replaced by "simulation"; trace is used here for generality because a dump file can be generated by several tools.

exist before the end of the trace. There are no value changes in between the two marking the beginning and end (if they exist); a move to the next value change from the start point leads to the end point.

30.8.5 Sample code using object (and traverse) collections

```
p_vpi_extension reader;   /* Pointer to reader VPI library */
vpiHandle scope;          /* Some scope being looked at     */
vpiHandle var_handle;     /* Object handle                  */
vpiHandle some_net;       /* Handle of some net             */
vpiHandle some_reg;       /* Handle of some reg             */
vpiHandle vc_trvs_hdl1;   /* Traverse handle                */
vpiHandle vc_trvs_hdl2;   /* Traverse handle                */
vpiHandle itr;            /* Iterator                       */
vpiHandle objCollection;  /* Object collection              */
vpiHandle trvsCollection; /* Traverse collection            */

PLI_BYTE8 *data = "my_database";/* database                 */
p_vpi_time time_p;              /* time                      */
PLI_INT32 code;           /* Return code                    */

/* Initialize the read interface: Post-process mode, read from a database */
/* NOTE: Uses "toolX" library                                             */
reader_p = vpi_load_extension("toolX", data, vpiAccessPostProcess);

if (reader_p == NULL) ... ; /* Not successful */

/* Get the scope using its name */
scope = reader_p->vpi_handle_by_name("top.m1.s1", NULL);
/* Create object collection */
objCollection = reader_p->vpi_create(vpiObjCollection, NULL, NULL);

/* Add data to collection: All the nets in scope */
/* ASSUMPTION: (waveform) tool "toolX" supports this navigation
   relationship */
itr = reader_p->vpi_iterate(vpiNet, scope);
while (var_handle = reader_p->vpi_scan(itr)) {
   objCollection = reader_p->vpi_create(vpiObjCollection, objCollection,
   var_handle);
}
/* Add data to collection: All the regs in scope */
/* ASSUMPTION: (waveform) tool supports this navigation relationship */
itr = reader_p->vpi_iterate(vpiReg, scope);
while (var_handle = reader_p->vpi_scan(itr)) {
   objCollection = reader_p->vpi_create(vpiObjCollection, objCollection,
   var_handle);
}

/* Initialize the load: focus only on the signals in the object collection:
objCollection */
reader_p->vpi_load_init(objCollection, NULL, 0);

/* Demo scanning the object collection */
itr = reader_p->vpi_iterate(vpiMember, objCollection);
while (var_handle = reader_p->vpi_scan(itr)) {
    ...
}

/* Application code here */
```

```
some_net = ...;
time_p = ...;
some_reg = ...;
....
vc_trvs_hdl1 = reader_p->vpi_handle(vpiTrvsObj, some_net);
vc_trvs_hdl2 = reader_p->vpi_handle(vpiTrvsObj, some_reg);
vc_trvs_hdl1 = reader_p->vpi_goto(vpiTime, vc_trvs_hdl1, time_p, &code);
vc_trvs_hdl2 = reader_p->vpi_goto(vpiTime, vc_trvs_hdl2, time_p, &code);
/* Data querying and processing here */
....

/* free handles */
reader_p->vpi_free_object(...);

/* close database */
reader_p->vpi_close(0, vpiAccessPostProcess, data);
```

The code segment above initializes the read interface for post-process read access from database data. It then creates an object collection `objCollection` and then adds to it all the objects in scope of type `vpiNet` and `vpiReg` (assuming this type of navigation is allowed in the tool). Load access is initialized and set to the objects listed in `objCollection`. `objCollection` can be iterated using `vpi_iterate()` to create the iterator and then using `vpi_scan()` to scan it assuming here that the waveform tool provides this navigation. The application code is then free to obtain traverse handles for the objects and perform its querying and data processing as it desires.

The code segment below shows a simple code segment that mimics the function of a `$dumpvars` call to access data of all the regs in a specific scope and its subscopes and process the data.

```
p_vpi_extension reader_p;  /* Reader library pointer        */
vpiHandle big_scope;       /* Some scope being looked at    */
vpiHandle obj_handle;      /* Object handle                 */
vpiHandle obj_trvs_hdl;    /* Traverse handle               */
vpiHandle signal_iterator; /* Iterator for signals          */
p_vpi_time time_p;         /* time                          */

/* Initialize the read interface: Access data from simulator        */
/* NOTE: Use built-in VPI (e.g., that of simulator application is running
under  */
reader_p = vpi_load_extension(NULL, NULL, vpiAccessLimitedInteractive);

if (reader_p == NULL) ... ; /* Not successful */

/* Initialize the load access: data from (simulator) memory, for scope
   big_scope and its subscopes */
/* NOTE: Call marks load access */
vpi_load_init(NULL, big_scope, 0);

/* Application code here */
/* Obtain handle for all the regs in scope */
signal_iterator = vpi_iterate(vpiReg, big_scope);

/* Data querying and processing here */
while ( (obj_handle = vpi_scan(signal_iterator)) != NULL ) {
   assert(vpi_get(vpiType, obj_handle) == vpiReg);
   /* Create a traverse handle for read queries */
   obj_trvs_hdl = vpi_handle(vpiTrvsObj, obj_handle);
   time_p = ...; /* some time */
   obj_trvs_hdl = vpi_goto(vpiTime, obj_trvs_hdl, time_p, &code);
```

```
    /* Get info at time */
    vpi_get_value(obj_trvs_hdl, value_p); /* Value */
    vpi_printf("....");
}
/* free handles*/
vpi_free_object(...);
```

30.8.6 Object-based traversal

Object-based traversal can be performed by creating a traverse handle for the object and then moving it back and forth to the next or previous value change or by performing jumps in time. A traverse object handle for any object in the design can be obtained by calling `vpi_handle()` with a `vpiTrvsObj` type and the object's handle. This is the method described in 30.8.4 and used in all the code examples thus far.

Using this method, the traversal would be object-based because the individual object traverse handles are created and then the application can query the (value, time) pairs for each value change. This method works well when the design is being navigated and there is a need to access the (stored) data of any individual object.

30.8.7 Time-ordered traversal

Alternatively, a user might wish to do a time-ordered traversal, i.e., a time-based examination of values of several objects. This can be done by using a collection. The first step is to create a traverse collection of type `vpiTrvsCollection` of the objects to be traversed from the design object collection of type `vpiObjCol-lection` using `vpi_handle()` with a `vpiTrvsCollection` type and collection handle argument. `vpi_goto()` can then be called on the traverse collection to move to next or previous or do jump in time for the collection as a whole. A move to next (previous) value change means move to the next (previous) earliest value change among the objects in the collection. Any handle that does not have a value change at that time is unchanged. A jump to a specific time aligns the new returned handles of all the objects in the collection (as if this had been done object by object, but here it is done in one shot for all elements).

It is possible to loop in time by incrementing the time and doing a jump to those time increments. This is shown in the following code snippet.

```
vpiHandle objCollection = ...;
vpiHandle trvsCollection;
p_vpi_time time_p;
PLI_INT32 code;

/* Obtain (create) traverse collection from object collection */
trvsCollection = vpi_handle(vpiTrvsCollection, objCollection);
/* Loop in time: increments of 100 units */
for (i = 0; i < 1000; i = i + 100) {
    time_p = ...;
    /* Go to point in time */
    trvsCollection = vpi_goto(vpiTime, trvsCollection, time_p, &code);
    ...
}
```

Alternatively, the user might wish to get a new collection returned of all the objects that have a value change at the given time to which the traverse collection was moved. In this case, `vpi_filter()` follows the call to `vpi_goto()`. The latter returns a new collection with all the new traverse objects, regardless of whether they have a value change. `vpi_filter()` allows filtering of the members that have a value change at that time. This is shown in the code snippet that follows.

```
...
```

```
vpiHandle rettrvsCollection; /* Collection for all the objects */
vpiHandle vctrvsCollection;  /* collection for the objects with VC */
vpiHandle itr;               /* collection member iterator */
...
/* Go to earliest next VC in the collection */
for (;;) { /* for all collection VCs in time */
   rettrvsCollection = vpi_goto(vpiNextVC, trvsCollection, NULL, &code);
   if (!code) {
      /* failure (e.g., already at MaxTime or no more VCs) */
      break; /* cannot go further */
   }
   vctrvsCollection = vpi_filter(rettrvsCollection, vpiHasVC, 1);
   /* create iterator then scan the VC collection */
   itr = vpi_iterate(vpiMember, vctrvsCollection);
   while (vc_trvs1_hdl = vpi_scan(itr)) {
      /* Element has a VC */
      vpi_get_value(vc_trvs1_hdl, value_p);  /* VC data */
      /* Do something at this VC point */
      ...
   }
   ...
}
```

30.9 Optionally unloading data

The implementation tool should handle unloading the unused data in a fashion invisible to the user. Managing the data caching and memory hierarchy is left to tool implementation, but it should be noted that failure to unload can affect the tool performance and capacity.

The user can optionally choose to call `vpi_unload()` to unload the data from (active) memory if the user application is done with accessing the data.

Calling `vpi_unload()` before releasing (freeing) traverse (collection) handles that are manipulating the data using `vpi_free_object()` is not recommended practice by users; the behavior of traversal using existing handles is not defined here. It is left up to tool implementation to decide how best to handle this. Tools shall, however, prevent creation of new traverse handles, after the call to `vpi_unload()`, by returning the appropriate fail codes in the respective creation routines.

30.10 Reading data from multiple databases and/or different read library providers

The VPI routine `vpi_load_extension()` is used to load VPI extensions. Such extensions include reader libraries from such tools as waveform viewers. `vpi_load_extension()` shall return a pointer to a function pointer structure with the following definition:

```
typedef struct {
   void *user_data;              /* Attach user data here if needed */
   /* Below this point user application MUST NOT modify any values */
   size_t struct_size;           /* Must be set to sizeof(s_vpi_extension) */
   long struct_version;          /* Set to 1 for SystemVerilog */
   PLI_BYTE8 *extension_version;
   PLI_BYTE8 *extension_name;
   /* One function pointer for each of the defined VPI routines:
      - Each function pointer has to have the correct prototype */
   ...
   PLI_INT32 (*vpi_chk_error)(error_info_p);
```

```
    ...
    PLI_INT32 (*vpi_vprintf)(PLI_BYTE8 *format, ...);
    ...
} s_vpi_extension, *p_vpi_extension;
```

Subsequent versions of the `s_vpi_extension` structure shall only extend it by adding members at the end; previously existing entries must not be changed, removed, or reordered in order to preserve backward compatability. The `struct_size` entry allows users to perform basic sanity checks (e.g., before type casting), and the `struct_version` permits keeping track and checking the version of the `s_vpi_extension` structure. The structure also has a `user_data` field to give users a way to attach data to a particular load of an extension if they wish to do so.

The structure shall have an entry for every VPI routine; the order and synopsis of these entries within the structure shall exactly match the order and synopsis of function definitions in Clause 27 of IEEE Std 1364. After those entries, the SystemVerilog VPI routine `vpi_register_assertion_cb()` shall be added. Then all new reader routines defined in Table 30-3 shall be added in exactly the order noted in the table. If a particular extension does not support a specific VPI routine, then it shall still have an entry (with the correct prototype) and a dummy body that shall always have a return (consistent with the VPI prototype) to signify failure (i.e., `NULL` or `FALSE`). The routine call must also raise the appropriate VPI error, which can be checked by `vpi_chk_error()`, and/or automatically generate an error message in a manner consistent with the specific VPI routine.

If tool providers want to add their own implementation extensions, those extensions must only have the effect of making the `s_vpi_extension` structure larger and any nonstandard content must occur after all the standard fields. This permits applications to use the pointer to the extended structure as if it was a `p_vpi_extension` pointer, yet still allow the applications to go beyond and access or call tool-specific fields or routines in the extended structure. For example, a tool-extended `s_vpi_extension` could be as follows:

```
typedef struct {
    /* inline a copy of s_vpi_extension                */
    /* begin                                           */
    void *user_data;
    ...
    /* end                                             */
    /* "toolZ" extension with one additional routine   */
    int (*toolZfunc)(int);
} s_toolZ_extension, *p_toolZ_extension;
```

An example of use of the above extended structure is as follows:

```
p_vpi_extension h;
p_toolZ_extension hZ;

h = vpi_load_extension("toolZ", <args>);
if ( h && (h->struct_size >= sizeof(s_toolZ_extension))
        && !(strcmp(h->extension_version, "...")
        && !strcmp(h->extension_name, "toolZ") ) {
    hZ = (p_toolZ_extension) h;
    /* Can now use hZ to access all the VPI routines, including toolZ's
       'toolZfunc' */
    ...
}
```

The SystemVerilog tool under which the user application is running is responsible for loading the appropriate extension, i.e., the reader API library in the case of the read API. The extension name is used for this

purpose, following a specific policy; for example, this extension name can be the name of the library to be loaded. Once the reader API library is loaded, all VPI function calls that wish to use the implementation in the library shall be performed using the returned `p_vpi_extension` pointer as an indirection to call the function pointers specified in `s_vpi_extension` or the extended vendor-specific structure as described above. As stated earlier, in any case that the application is using the built-in routine implementation (i.e., the ones provided by the tool, e.g., simulator, under which it is running), then the de-reference through the pointer is not necessary.

Multiple databases can be opened for read simultaneously by the application. After a `vpi_load_extension()` call, a top scope handle can be created for that database to be used later to derive any other handles for objects in that database. An example of multiple database access is shown below. In the example, `scope1` and `scope2` are the top scope handles used to point into `database1` and `database2`, respectively, and perform the processing (comparing data in the two databases, for example).

```
p_vpi_extension reader_pX; /* Pointer to reader libraryfunction struct */
p_vpi_extension reader_pY; /* Pointer to reader libraryfunction struct */
vpiHandle scope1, scope2;  /* Some scope being looked at              */
vpiHandle var_handle;      /* Object handle                          */
vpiHandle some_net;        /* Handle of some net                     */
vpiHandle some_reg;        /* Handle of some reg                     */
vpiHandle vc_trvs_hdl1;    /* Traverse handle                        */
vpiHandle vc_trvs_hdl2;    /* Traverse handle                        */
vpiHandle itr;             /* Iterator                               */
vpiHandle objCollection1, objCollection2;   /* Object collection      */
vpiHandle trvsCollection1, trvsCollection2; /* Traverse collection    */
p_vpi_time time_p;                          /* time                   */

PLI_BYTE8 *data1 = "database1";
PLI_BYTE8 *data2 = "database2";

/* Initialize the read interface: Post-process mode, read from a database */
/* NOTE: Use library from "toolX"                                 */
reader_pX = vpi_load_extension("toolX", data1, vpiAccessPostProcess);
/* Get the scope using its name */
/* NOTE: scope handle comes from database: data1                  */
scope1 = reader_pX->vpi_handle_by_name("top.m1.s1", NULL);

/* Initialize the read interface: Post-process mode, read from a database */
/* NOTE: Use library from "toolY"                                 */
reader_pY = vpi_load_extension("toolY", data2, vpiAccessPostProcess);
/* Get the scope using its name */
/* NOTE: scope handle comes from database: data2                  */
scope2 = reader_pY->vpi_handle_by_name("top.m1.s1", NULL);

/* Create object collections */
objCollection1 = reader_pX->vpi_create(vpiObjCollection, NULL, NULL);
objCollection2 = reader_pY->vpi_create(vpiObjCollection, NULL, NULL);

/* Add data to collection1: All the nets in scope1,
   data comes from database1 */
/* ASSUMPTION: (waveform) tool supports this navigation relationship */
itr = reader_pX->vpi_iterate(vpiNet, scope1);
while (var_handle = reader_pX->vpi_scan(itr)) {
   objCollection1 = reader_pX->vpi_create(vpiObjCollection, objCollection1,
   var_handle);
}
```

```
/* Add data to collection2: All the nets in scope2,
   data comes from database2 */
/* ASSUMPTION: (waveform) tool supports this navigation relationship */
itr = reader_pY->vpi_iterate(vpiNet, scope2);
while (var_handle = reader_pY->vpi_scan(itr)) {
   objCollection2 = reader_pY->vpi_create(vpiObjCollection, objCollection2,
   var_handle);
}

/* Initialize the load: focus only on the signals in the object collection:
objCollection */
reader_pX->vpi_load_init(objCollection1, NULL, 0);
reader_pY->vpi_load_init(objCollection2, NULL, 0);

/* Demo: Scan the object collection */
itr = reader_pX->vpi_iterate(vpiMember, objCollection1);
while (var_handle = reader_pX->vpi_scan(itr)) {
   ...
}
itr = reader_pY->vpi_iterate(vpiMember, objCollection2);
while (var_handle = reader_pY->vpi_scan(itr)) {
   ...
}

/* Application code here: Access Objects from database1 or database2 */
some_net = ...;
time_p = ...;
some_reg = ...;
....
/* Data querying and processing here */
....

/* free handles*/
reader_pX->vpi_free_object(...);
reader_pY->vpi_free_object(...);


/* close databases */
reader_pX->vpi_close(0, vpiAccessPostProcess, data1);
reader_pY->vpi_close(0, vpiAccessPostProcess, data2);
```

30.11 VPI routines extended in SystemVerilog

Table 30-1 lists the usage extensions. They are repeated here as the additional extended usage with traverse
(collection) handles of vpi_get_time() for clarity.

```
vpi_get_time()
```
Synopsis: Retrieve the time of the object or collection of objects traverse handle.
Syntax: vpi_get_time(vpiHandle obj, p_vpi_time time_p)
Returns: void
Arguments:

    vpiHandle obj: Handle to a traverse object of type vpiTrvsObj or a traverse collection of
    type vpiTrvsCollection.

    p_vpi_time time_p: Pointer to a structure containing the returned time information. There are
    several cases to consider:

        PLI_INT32 type = ...; /* vpiScaledRealTime, vpiSimTime, or vpiSuppressTime */
        (time_p == type): Get the time of traverse object or collection. In case of a collection,

return time only if all the members have the same time; otherwise, `time_p` is not modified.

(`time_p == vpiMinTime & type`): Gets the minimum time of traverse object or collection.

(`time_p == vpiMaxTime & type`): Gets the maximum time of traverse object or collection.

(`time_p == vpiNextVC & type`): Gets the time where traverse handle points next. Returns failure if traverse object or collection has no next value change and `time_p` is not modified. In the case of a collection, it returns success when any traverse object in the collection has a next value change; `time_p` is updated with the smallest next value change time.

(`time_p == vpiPrevVC & type`): Gets the time where traverse handle previously points. Returns failure if traverse object or collection has no previous value change and `time_p` is not modified. In the case of a collection, it returns success when any traverse object in the collection has a previous value change; `time_p` is updated with the largest previous value change time.

Related routines: None.

30.12 VPI routines added in SystemVerilog

This subclause describes the additional VPI routines in detail.

`vpi_load_extension()`
Synopsis: Load-specified VPI extension. The general form of this function allows for later extensions. For the reader-specific form, initialize the reader with access mode, and specify the database if used.
Syntax: `vpi_load_extension( PLI_BYTE8 *extension_name, ...)` in its general form
`vpi_load_extension( PLI_BYTE8 *extension_name,`
`PLI_BYTE8 *name,`
`PLI_INT32 mode, ...)` for the reader extension
Returns: `PLI_INT32`, 1 for success, 0 for fail.
Arguments:
  `PLI_BYTE8 *extension_name`: Extension name of the extension library to be loaded. In the case of the reader, this is the reader VPI library (with the supported navigation VPI routines).
  `...`: Contains all the additional arguments. For the reader extension, these are as follows:
  `PLI_BYTE8 *name`: Database.
  `PLI_INT32 mode`:
    `vpiAccessLimitedInteractive`: Access data in tool memory, with limited history. The tool shall at least have the current time value; no history is required.
    `vpiAccessInteractive`: Access data interactively. Tool shall keep value history up to the current time.
    `vpiAccessPostProcess`: Access data stored in specified database.
  `...`: Additional arguments if required by specific reader extensions.
Related routines: None.

30.12.1 VPI reader routines

`vpi_close()`
Synopsis: Close the database if open.
Syntax: `vpi_close(PLI_INT32 tool, PLI_INT32 prop, PLI_BYTE8* name)`
Returns: `PLI_INT32`, 1 for success, 0 for fail.
Arguments:
  `PLI_INT32 tool`: 0 for the reader.
  `PLI_INT32 prop`:
    `vpiAccessPostProcess`: Access data stored in specified database.

vpiAccessInteractive: Access data interactively; database is the flush area. Tool shall keep value history up to the current time.

PLI_BYTE8* name: Name of the database. This can be the logical name of a database or the actual name of the data file depending on the tool implementation.

Related routines: None.

vpi_load_init()

Synopsis: Initialize the load access to scope and/or collection of objects.

Syntax: vpi_load_init(vpiHandle objCollection, vpiHandle scope, PLI_INT32 level)

Returns: PLI_INT32, 1 for success, 0 for fail.

Arguments:

vpiHandle objCollection: Object collection of type vpiObjCollection, a collection of design objects.

vpiHandle scope: Scope of the load.

PLI_INT32 level: If 0, then enables read access to scope and all its subscopes; 1 means just the scope.

Related routines: None.

vpi_load()

Synopsis: Load the data of the given object into memory for data access and traversal if object is an object handle; load the whole collection (i.e., set of objects) if passed handle is an object collection of type vpiObjCollection.

Syntax: vpi_load(vpiHandle h)

Returns: PLI_INT32, 1 for success of loading (all) object(s) (in collection), 0 for fail of loading (any) object (in collection).

Arguments:

vpiHandle h: Handle to a design object (of any valid type) or object collection of type vpiObjCollection.

Related routines: None

vpi_unload()

Synopsis: Unload the given object data from (active) memory if object is an object handle; unload the whole collection if passed object is a collection of type vpiObjCollection. See 30.9 for a description of data unloading.

Syntax: vpi_unload(vpiHandle h)

Returns: PLI_INT32, 1 for success, 0 for fail.

Arguments:

vpiHandle h: Handle to an object or collection (of type vpiObjCollection).

Related routines: None.

vpi_create()

Synopsis: Create or add to an object or traverse collection.

Syntax: vpi_create(PLI_INT32 prop, vpiHandle h, vpiHandle obj)

Returns: vpiHandle of type vpiObjCollection for success, NULL for fail.

Arguments:

PLI_INT32 prop:

vpiObjCollection: Create (or add to) object (vpiObjCollection) or traverse (vpiTrvsCollection) collection.

vpiHandle h: Handle to a (object) traverse collection of type (vpiObjCollection) vpiTrvsCollection, NULL for first call (creation)

vpiHandle obj: Handle of object to add, NULL if for first time creation of collection.

Related routines: None.

vpi_goto()

Synopsis: Try to move to minimum, maximum, or specified time. A new traverse (collection) handle is returned pointing to the specified time. If the traverse handle (members of collection) has a value change at that time, then the returned handle (members of returned collection) is updated to point to the

specified time; otherwise, it is not updated. If the passed handle has no value change (for collection this means no value change for any object), a fail is indicated; otherwise, a success is indicated. In the case of a jump to a specified time and no value change at the specified time, then the value change traverse index of the returned (new) handle (member of returned collection) is aligned based on the jump behavior defined in 30.8.4.2, and its time (and the time pointer argument if passed and non-NULL) shall be updated based on the aligned traverse point. In the case of vpiNextVC or vpiPrevVC, the time argument, if passed and non-NULL (otherwise, it is ignored and not updated), is updated if there is a value change (for collection this means a value change for any object) to the new time; otherwise, the value is not updated.

Syntax: vpi_goto(PLI_INT32 prop, vpiHandle obj, p_vpi_time time_p, PLI_INT32 *ret_code)

Returns: vpiHandle of type vpitrvsObj (vpiObjCollection).

Arguments:

PLI_INT32 prop:

    vpiMinTime: Goto the minimum time of traverse collection handle.

    vpiMaxTime: Goto the maximum time of traverse collection handle.

    vpiTime: Jump to the time specified in time_p.

    vpiNextVC: Goto the (time of) next value change.

    vpiPrevVC: Goto the (time of) previous value change.

vpiHandle obj: Handle to a traverse object (collection) of type vpiTrvsObj (vpiTrvsCollection)

p_vpi_time time_p: Pointer to a structure containing time information. Used only if prop is of type vpiTime; otherwise, it is ignored.

PLI_INT32 *ret_code: Pointer to a return code indicator. It is 1 for success and 0 for fail.

Related routines: None.

vpi_filter()

Synopsis: Filter a general collection, a traversable object collection, or traverse collection according to a specific criterion. Return a collection of the handles that meet the criterion. Original collection is not changed.

Syntax: vpi_filter(vpiHandle h, PLI_INT32 ft, PLI_INT32 flag)

Returns: vpiHandle of type vpiObjCollection for success, NULL for fail.

Arguments:

vpiHandle h: Handle to a collection of type vpiCollection, vpiObjCollection or vpiTrvsCollection

PLI_INT32 ft: Filter criterion, any vpiType or a VPI boolean property.

PLI_INT32 flag: Flag to indicate whether to match criterion (if set to TRUE) or not (if set to FALSE).

Related routines: None.

Annex A

(normative)

Formal syntax

The formal syntax of SystemVerilog is described using Backus-Naur Form (BNF). The syntax of System-Verilog HDL source is derived from the starting symbol source_text. The syntax of a library map file is derived from the starting symbol library_text. The conventions used are as follows:

— Keywords and punctuation are in **bold-red** text.
— Syntactic categories are named in nonbold text.
— A vertical bar (|) separates alternatives.
— Square brackets ([ ]) enclose optional items.
— Braces ({ }) enclose items that can be repeated zero or more times.

The full syntax and semantics of Verilog and SystemVerilog are not described solely using BNF. The normative text description contained within the chapters of IEEE Std 1364 and this standard provide additional details on the syntax and semantics described in this BNF.

A.1 Source text

A.1.1 Library source text

library_text ::= { library_description }
library_description ::=
      library_declaration
    | include_statement
    | config_declaration
    | **;**
library_declaration ::=
      library library_identifier file_path_spec { **,** file_path_spec }
        [**-incdir** file_path_spec { **,** file_path_spec }] **;**
include_statement ::= **include** file_path_spec **;**

A.1.2 SystemVerilog source text

source_text ::= [timeunits_declaration] { description }
description ::=
      module_declaration
    | udp_declaration
    | interface_declaration
    | program_declaration
    | package_declaration
    | { attribute_instance } package_item
    | { attribute_instance } bind_directive
    | config_declaration
module_nonansi_header ::=
      { attribute_instance } module_keyword [lifetime] module_identifier [parameter_port_list]

list_of_ports **;**

module_ansi_header ::=
 { attribute_instance } module_keyword [lifetime] module_identifier [parameter_port_list]
  [list_of_port_declarations] **;**

module_declaration ::=
 module_nonansi_header [timeunits_declaration] { module_item }
  endmodule [**:** module_identifier]
 | module_ansi_header [timeunits_declaration] { non_port_module_item }
  endmodule [**:** module_identifier]
 | { attribute_instance } module_keyword [lifetime] module_identifier **(.*) ;**
  [timeunits_declaration] { module_item } **endmodule** [**:** module_identifier]
 | **extern** module_nonansi_header
 | **extern** module_ansi_header

module_keyword ::= **module** | **macromodule**

interface_nonansi_header ::=
 { attribute_instance } **interface** [lifetime] interface_identifier
  [parameter_port_list] list_of_ports **;**

interface_ansi_header ::=
 {attribute_instance } **interface** [lifetime] interface_identifier
  [parameter_port_list] [list_of_port_declarations] **;**

interface_declaration ::=
 interface_nonansi_header [timeunits_declaration] { interface_item }
  endinterface [**:** interface_identifier]
 | interface_ansi_header [timeunits_declaration] { non_port_interface_item }
  endinterface [**:** interface_identifier]
 | { attribute_instance } **interface** interface_identifier **(.*) ;**
  [timeunits_declaration] { interface_item }
 endinterface [**:** interface_identifier]
 | **extern** interface_nonansi_header
 | **extern** interface_ansi_header

program_nonansi_header ::=
 { attribute_instance } **program** [lifetime] program_identifier
  [parameter_port_list] list_of_ports **;**

program_ansi_header ::=
 {attribute_instance } **program** [lifetime] program_identifier
  [parameter_port_list] [list_of_port_declarations] **;**

program_declaration ::=
 program_nonansi_header [timeunits_declaration] { program_item }
  endprogram [**:** program_identifier]
 | program_ansi_header [timeunits_declaration] { non_port_program_item }
  endprogram [**:** program_identifier]
 | { attribute_instance } **program** program_identifier **(.*) ;**
  [timeunits_declaration] { program_item }
 endprogram [**:** program_identifier]
 | **extern** program_nonansi_header
 | **extern** program_ansi_header

class_declaration ::=
 [**virtual**] **class** [lifetime] class_identifier [parameter_port_list]
  [**extends** class_type [**(** list_of_arguments **)**]]**;**
  { class_item }
 endclass [**:** class_identifier]

package_declaration ::=
    { attribute_instance } **package** package_identifier **;**
      [timeunits_declaration] { { attribute_instance } package_item }
    endpackage [**:** package_identifier]

timeunits_declaration ::=
    timeunit time_literal **;**
    | **timeprecision** time_literal **;**
    | **timeunit** time_literal **;**
    timeprecision time_literal **;**
    | **timeprecision** time_literal **;**
    timeunit time_literal **;**

A.1.3 Module parameters and ports

parameter_port_list ::=
    # (list_of_param_assignments { **,** parameter_port_declaration } **)**
    | **# (** parameter_port_declaration { **,** parameter_port_declaration } **)**
    | **#( )**

parameter_port_declaration ::=
    parameter_declaration
    | data_type list_of_param_assignments
    | **type** list_of_type_assignments

list_of_ports ::= **(** port { **,** port } **)**

list_of_port_declarations[25] ::=
    ([{ attribute_instance} ansi_port_declaration { **,** { attribute_instance} ansi_port_declaration }] **)**

port_declaration ::=
    { attribute_instance } inout_declaration
    | { attribute_instance } input_declaration
    | { attribute_instance } output_declaration
    | { attribute_instance } ref_declaration
    | { attribute_instance } interface_port_declaration

port ::=
    [port_expression]
    | **.** port_identifier **(** [port_expression] **)**

port_expression ::=
    port_reference
    | **{** port_reference { **,** port_reference } **}**

port_reference ::=
    port_identifier constant_select

port_direction ::= **input** | **output** | **inout** | **ref**

net_port_header ::= [port_direction] net_port_type

variable_port_header ::= [port_direction] variable_port_type

interface_port_header ::=
    interface_identifier [**.** modport_identifier]
    | **interface** [**.** modport_identifier]

ansi_port_declaration ::=
    [net_port_header | interface_port_header] port_identifier { unpacked_dimension }
    | [variable_port_header] port_identifier { variable_dimension } [**=** constant_expression]
    | [net_port_header | variable_port_header] **.** port_identifier **(** [expression] **)**

513

A.1.4 Module items

module_common_item ::=
      module_or_generate_item_declaration
    | interface_instantiation
    | program_instantiation
    | concurrent_assertion_item
    | bind_directive
    | continuous_assign
    | net_alias
    | initial_construct
    | final_construct
    | always_construct
    | loop_generate_construct
    | conditional_generate_construct

module_item ::=
      port_declaration **;**
    | non_port_module_item

module_or_generate_item ::=
      { attribute_instance } parameter_override
    | { attribute_instance } gate_instantiation
    | { attribute_instance } udp_instantiation
    | { attribute_instance } module_instantiation
    | { attribute_instance } module_common_item

module_or_generate_item_declaration ::=
      package_or_generate_item_declaration
    | genvar_declaration
    | clocking_declaration
    | **default clocking** clocking_identifier **;**

non_port_module_item ::=
      generate_region
    | module_or_generate_item
    | specify_block
    | { attribute_instance } specparam_declaration
    | program_declaration
    | module_declaration
    | interface_declaration
    | timeunits_declaration[17]

parameter_override ::= **defparam** list_of_defparam_assignments **;**

bind_directive ::=
      bind bind_target_scope [**:** bind_target_instance_list] bind_instantiation **;**
    | **bind** bind_target_instance bind_instantiation **;**

bind_target_scope ::=
      module_identifier
    | interface_identifier

bind_target_instance ::=
      hierarchical_identifier constant_bit_select

bind_target_instance_list ::=
      bind_target_instance { **,** bind_target_instance }

bind_instantiation ::=
      program_instantiation

    | module_instantiation
    | interface_instantiation

A.1.5 Configuration source text

config_declaration ::=
    config config_identifier **;**
      design_statement
      { config_rule_statement }
    endconfig [**:** config_identifier]

design_statement ::= **design** { [library_identifier **.**] cell_identifier } **;**

config_rule_statement ::=
    default_clause liblist_clause **;**
    | inst_clause liblist_clause **;**
    | inst_clause use_clause **;**
    | cell_clause liblist_clause **;**
    | cell_clause use_clause **;**

default_clause ::= **default**

inst_clause ::= **instance** inst_name

inst_name ::= topmodule_identifier { **.** instance_identifier }

cell_clause ::= **cell** [library_identifier **.**] cell_identifier

liblist_clause ::= **liblist** {library_identifier}

use_clause ::= **use** [library_identifier **.**] cell_identifier [**: config**]

A.1.6 Interface items

interface_or_generate_item ::=
    { attribute_instance } module_common_item
    | { attribute_instance } modport_declaration
    | { attribute_instance } extern_tf_declaration

extern_tf_declaration ::=
    extern method_prototype **;**
    | **extern forkjoin** task_prototype **;**

interface_item ::=
    port_declaration **;**
    | non_port_interface_item

non_port_interface_item ::=
    generate_region
    | interface_or_generate_item
    | program_declaration
    | interface_declaration
    | timeunits_declaration[17]

A.1.7 Program items

program_item ::=
    port_declaration **;**
    | non_port_program_item

non_port_program_item ::=

```
        { attribute_instance } continuous_assign
      | { attribute_instance } module_or_generate_item_declaration
      | { attribute_instance } initial_construct
      | { attribute_instance } final_construct
      | { attribute_instance } concurrent_assertion_item
      | { attribute_instance } timeunits_declaration¹⁷
      | program_generate_item

program_generate_item³⁷ ::=
        loop_generate_construct
      | conditional_generate_construct
      | generate_region
```

A.1.8 Class items

```
class_item ::=
        { attribute_instance } class_property
      | { attribute_instance } class_method
      | { attribute_instance } class_constraint
      | { attribute_instance } class_declaration
      | { attribute_instance } timeunits_declaration¹⁷
      | { attribute_instance } covergroup_declaration
      | ;
class_property ::=
        { property_qualifier } data_declaration
      | const { class_item_qualifier } data_type const_identifier [ = constant_expression ] ;
class_method ::=
        { method_qualifier } task_declaration
      | { method_qualifier } function_declaration
      | extern { method_qualifier } method_prototype ;
      | { method_qualifier } class_constructor_declaration
      | extern { method_qualifier } class_constructor_prototype
class_constructor_prototype ::=
        function new ( [ tf_port_list ] ) ;
class_constraint ::=
        constraint_prototype
      | constraint_declaration
class_item_qualifier⁷ ::=
        static
      | protected
      | local
property_qualifier⁷ ::=
        random_qualifier
      | class_item_qualifier
random_qualifier⁷ ::=
        rand
      | randc
method_qualifier⁷ ::=
        virtual
      | class_item_qualifier
```

method_prototype ::=
      task_prototype
    | function_prototype

class_constructor_declaration ::=
      function [class_scope] **new** [**(** [tf_port_list] **)**] **;**
        { block_item_declaration }
        [**super . new** [**(** list_of_arguments **)**] **;**]
        { function_statement_or_null }
      endfunction [**: new**]

A.1.9 Constraints

constraint_declaration ::= [**static**] **constraint** constraint_identifier constraint_block

constraint_block ::= **{** { constraint_block_item } **}**

constraint_block_item ::=
      solve identifier_list **before** identifier_list **;**
    | constraint_expression

constraint_expression ::=
      expression_or_dist **;**
    | expression **–>** constraint_set
    | **if (** expression **)** constraint_set [**else** constraint_set]
    | **foreach (** array_identifier **[** loop_variables **])** constraint_set

constraint_set ::=
      constraint_expression
    | **{** { constraint_expression } **}**

dist_list ::= dist_item { **,** dist_item }

dist_item ::= value_range [dist_weight]

dist_weight ::=
      := expression
    | **:/** expression

constraint_prototype ::= [**static**] **constraint** constraint_identifier **;**

extern_constraint_declaration ::=
      [**static**] **constraint** class_scope constraint_identifier constraint_block

identifier_list ::= identifier { **,** identifier }

A.1.10 Package items

package_item ::=
      package_or_generate_item_declaration
    | anonymous_program
    | timeunits_declaration[17]

package_or_generate_item_declaration ::=
      net_declaration
    | data_declaration
    | task_declaration
    | function_declaration
    | dpi_import_export
    | extern_constraint_declaration
    | class_declaration

```
        | class_constructor_declaration
        | parameter_declaration ;
        | local_parameter_declaration
        | covergroup_declaration
        | overload_declaration
        | concurrent_assertion_item_declaration
        | ;
anonymous_program ::= program ; { anonymous_program_item } endprogram
anonymous_program_item ::=
        task_declaration
        | function_declaration
        | class_declaration
        | covergroup_declaration
        | class_constructor_declaration
        | ;
```

A.2 Declarations

A.2.1 Declaration types

A.2.1.1 Module parameter declarations

```
local_parameter_declaration ::=
        localparam data_type_or_implicit  list_of_param_assignments ;
        | localparam type  list_of_type_assignments ;
parameter_declaration ::=
        parameter data_type_or_implicit  list_of_param_assignments
        | parameter type  list_of_type_assignments
specparam_declaration ::=
        specparam [ packed_dimension ] list_of_specparam_assignments ;
```

A.2.1.2 Port declarations

```
inout_declaration ::=
        inout net_port_type list_of_port_identifiers
input_declaration ::=
        input net_port_type list_of_port_identifiers
        | input variable_port_type list_of_variable_identifiers
output_declaration ::=
        output net_port_type list_of_port_identifiers
        | output variable_port_type  list_of_variable_port_identifiers
interface_port_declaration ::=
        interface_identifier list_of_interface_identifiers
        | interface_identifier . modport_identifier  list_of_interface_identifiers
ref_declaration ::= ref variable_port_type list_of_port_identifiers
```

A.2.1.3 Type declarations

```
data_declaration14 ::=
        [ const ] [ var ] [ lifetime ] data_type_or_implicit list_of_variable_decl_assignments ;
```

    | type_declaration
    | package_import_declaration
    | virtual_interface_declaration

package_import_declaration ::=
    import package_import_item { **,** package_import_item } **;**

package_import_item ::=
    package_identifier **::** identifier
    | package_identifier **:: ***

genvar_declaration ::= **genvar** list_of_genvar_identifiers **;**

net_declaration[13] ::=
    net_type [drive_strength | charge_strength] [**vectored** | **scalared**]
      data_type_or_implicit [delay3] list_of_net_decl_assignments **;**

type_declaration ::=
    typedef data_type type_identifier { variable_dimension } **;**
    | **typedef** interface_instance_identifier **.** type_identifier type_identifier **;**
    | **typedef** [**enum** | **struct** | **union** | **class**] type_identifier **;**

lifetime ::= **static** | **automatic**

A.2.2 Declaration data types

A.2.2.1 Net and variable types

casting_type ::= simple_type | constant_primary | signing

data_type ::=
    integer_vector_type [signing] { packed_dimension }
    | integer_atom_type [signing]
    | non_integer_type
    | struct_union [**packed** [signing]] **{** struct_union_member { struct_union_member } **}**
      { packed_dimension }[12]
    | **enum** [enum_base_type] **{** enum_name_declaration { **,** enum_name_declaration } **}**
    | **string**
    | **chandle**
    | **virtual** [**interface**] interface_identifier
    | [class_scope | package_scope] type_identifier { packed_dimension }
    | class_type
    | **event**
    | ps_covergroup_identifier
    | type_reference[28]

data_type_or_implicit ::=
    data_type
    | [signing] { packed_dimension }

enum_base_type ::=
    integer_atom_type [signing]
    | integer_vector_type [signing] [packed_dimension]
    | type_identifier [packed_dimension][23]

enum_name_declaration ::=
    enum_identifier [**[** integral_number [**:** integral_number] **]**] [**=** constant_expression]

class_scope ::= class_type **::**

class_type ::=

ps_class_identifier [parameter_value_assignment]
    { **::** class_identifier [parameter_value_assignment] }

integer_type ::= integer_vector_type | integer_atom_type

integer_atom_type ::= **byte** | **shortint** | **int** | **longint** | **integer** | **time**

integer_vector_type ::= **bit** | **logic** | **reg**

non_integer_type ::= **shortreal** | **real** | **realtime**

net_type ::= **supply0** | **supply1** | **tri** | **triand** | **trior** | **trireg**| **tri0** | **tri1** | **uwire**| **wire** | **wand** | **wor**

net_port_type[33] ::=
    [net_type] data_type_or_implicit

variable_port_type ::= var_data_type

var_data_type ::= data_type | **var** data_type_or_implicit

signing ::= **signed** | **unsigned**

simple_type ::= integer_type | non_integer_type | ps_type_identifier | ps_parameter_identifier

struct_union_member[26] ::=
    { attribute_instance } [random_qualifier] data_type_or_void list_of_variable_decl_assignments **;**

data_type_or_void ::= data_type | **void**

struct_union ::= **struct** | **union** [**tagged**]

type_reference ::=
    type (expression[27] **)**
   | **type (** data_type **)**

A.2.2.2 Strengths

drive_strength ::=
    (strength0 , strength1 **)**
   | **(** strength1 , strength0 **)**
   | **(** strength0 , **highz1)**
   | **(** strength1 , **highz0)**
   | **(highz0** , strength1 **)**
   | **(highz1** , strength0 **)**

strength0 ::= **supply0** | **strong0** | **pull0** | **weak0**

strength1 ::= **supply1** | **strong1** | **pull1** | **weak1**

charge_strength ::= **(small)** | **(medium)** | **(large)**

A.2.2.3 Delays

delay3 ::= **#** delay_value | **# (** mintypmax_expression [**,** mintypmax_expression [**,** mintypmax_expression]
   ] **)**

delay2 ::= **#** delay_value | **# (** mintypmax_expression [**,** mintypmax_expression] **)**

delay_value ::=
    unsigned_number
   | real_number
   | ps_identifier
   | time_literal

A.2.3 Declaration lists

list_of_defparam_assignments ::= defparam_assignment { **,** defparam_assignment }

list_of_genvar_identifiers ::= genvar_identifier { **,** genvar_identifier }

list_of_interface_identifiers ::= interface_identifier { unpacked_dimension }
       { **,** interface_identifier { unpacked_dimension } }

list_of_net_decl_assignments ::= net_decl_assignment { **,** net_decl_assignment }

list_of_param_assignments ::= param_assignment { **,** param_assignment }

list_of_port_identifiers ::= port_identifier { unpacked_dimension }
       { **,** port_identifier { unpacked_dimension } }

list_of_udp_port_identifiers ::= port_identifier { **,** port_identifier }

list_of_specparam_assignments ::= specparam_assignment { **,** specparam_assignment }

list_of_tf_variable_identifiers ::= port_identifier { variable_dimension } [**=** expression]
       { **,** port_identifier { variable_dimension } [**=** expression] }

list_of_type_assignments ::= type_assignment { **,** type_assignment }

list_of_variable_decl_assignments ::= variable_decl_assignment { **,** variable_decl_assignment }

list_of_variable_identifiers ::= variable_identifier { variable_dimension }
       { **,** variable_identifier { variable_dimension } }

list_of_variable_port_identifiers ::= port_identifier { variable_dimension } [**=** constant_expression]
       { **,** port_identifier { variable_dimension } [**=** constant_expression] }

list_of_virtual_interface_decl ::=
    variable_identifier [**=** interface_instance_identifier]
       { **,** variable_identifier [**=** interface_instance_identifier] }

A.2.4 Declaration assignments

defparam_assignment ::= hierarchical_parameter_identifier **=** constant_mintypmax_expression

net_decl_assignment ::= net_identifier { unpacked_dimension } [**=** expression]

param_assignment ::= parameter_identifier { unpacked_dimension } **=** constant_param_expression

specparam_assignment ::=
    specparam_identifier **=** constant_mintypmax_expression
  | pulse_control_specparam

type_assignment ::=
    type_identifier **=** data_type

pulse_control_specparam ::=
    PATHPULSE$ = (reject_limit_value [**,** error_limit_value] **)**
  | **PATHPULSE$**specify_input_terminal_descriptor**$**specify_output_terminal_descriptor
    = (reject_limit_value [**,** error_limit_value] **)**

error_limit_value ::= limit_value

reject_limit_value ::= limit_value

limit_value ::= constant_mintypmax_expression

variable_decl_assignment ::=
    variable_identifier { variable_dimension } [**=** expression]
  | dynamic_array_variable_identifier **[ ]** [**=** dynamic_array_new]
  | class_variable_identifier [**=** class_new]
  | [covergroup_variable_identifier] **= new** [**(** list_of_arguments **)**][15]

class_new[19] ::= **new** [**(** list_of_arguments **)** | expression]

dynamic_array_new ::= **new [** expression **]** [**(** expression **)**]

A.2.5 Declaration ranges

unpacked_dimension ::= **[** constant_range **]**
    | **[** constant_expression **]**

packed_dimension[11] ::=
      [constant_range **]**
    | unsized_dimension

associative_dimension ::=
      [data_type **]**
    | **[*]**

variable_dimension ::=
      unsized_dimension
    | unpacked_dimension
    | associative_dimension
    | queue_dimension

queue_dimension ::= **[$** [**:** constant_expression] **]**

unsized_dimension ::= **[ ]**

A.2.6 Function declarations

function_data_type ::= data_type | **void**

function_data_type_or_implicit ::=
      function_data_type
    | [signing] { packed_dimension }

function_declaration ::= **function** [lifetime] function_body_declaration

function_body_declaration ::=
      function_data_type_or_implicit
        [interface_identifier **.** | class_scope] function_identifier **;**
      { tf_item_declaration }
      { function_statement_or_null }
      endfunction [**:** function_identifier]
    | function_data_type_or_implicit
        [interface_identifier **.** | class_scope] function_identifier **(** [tf_port_list] **) ;**
      { block_item_declaration }
      { function_statement_or_null }
      endfunction [**:** function_identifier]

function_prototype ::= **function** function_data_type function_identifier **(** [tf_port_list] **)**

dpi_import_export ::=
      import dpi_spec_string [dpi_function_import_property] [c_identifier **=**] dpi_function_proto **;**
    | **import** dpi_spec_string [dpi_task_import_property] [c_identifier **=**] dpi_task_proto **;**
    | **export** dpi_spec_string [c_identifier **=**] **function** function_identifier **;**
    | **export** dpi_spec_string [c_identifier =] **task** task_identifier **;**

dpi_spec_string ::= **"DPI-C"** | **"DPI"**

dpi_function_import_property ::= **context** | **pure**

dpi_task_import_property ::= **context**

dpi_function_proto[8,9] ::= function_prototype

dpi_task_proto[9] ::= task_prototype

A.2.7 Task declarations

task_declaration ::= **task** [lifetime] task_body_declaration

task_body_declaration ::=
        [interface_identifier **.** | class_scope] task_identifier **;**
        { tf_item_declaration }
        { statement_or_null }
        endtask [**:** task_identifier]
    | [interface_identifier **.** | class_scope] task_identifier **(** [tf_port_list] **) ;**
        { block_item_declaration }
        { statement_or_null }
        endtask [**:** task_identifier]

tf_item_declaration ::=
        block_item_declaration
    | tf_port_declaration

tf_port_list ::=
        tf_port_item { **,** tf_port_item }

tf_port_item[34] ::=
        { attribute_instance }
            [tf_port_direction] [**var**] data_type_or_implicit
            [port_identifier { variable_dimension } [**=** expression]]

tf_port_direction ::= port_direction | **const ref**

tf_port_declaration ::=
        { attribute_instance } tf_port_direction [**var**] data_type_or_implicit list_of_tf_variable_identifiers **;**

task_prototype ::= **task** task_identifier **(** [tf_port_list] **)**

A.2.8 Block item declarations

block_item_declaration ::=
        { attribute_instance } data_declaration
    | { attribute_instance } local_parameter_declaration
    | { attribute_instance } parameter_declaration **;**
    | { attribute_instance } overload_declaration

overload_declaration ::=
        bind overload_operator **function** data_type function_identifier **(** overload_proto_formals **) ;**

overload_operator ::= **+** | **++** | **–** | **– –** | ***** | ****** | **/** | **%** | **==** | **!=** | **<** | **<=** | **>** | **>=** | **=**

overload_proto_formals ::= data_type {**,** data_type}

A.2.9 Interface declarations

virtual_interface_declaration ::=
        virtual [**interface**] interface_identifier list_of_virtual_interface_decl **;**

modport_declaration ::= **modport** modport_item { **,** modport_item } **;**

modport_item ::= modport_identifier **(** modport_ports_declaration { **,** modport_ports_declaration } **)**

modport_ports_declaration ::=
        { attribute_instance } modport_simple_ports_declaration
    | { attribute_instance } modport_tf_ports_declaration
    | { attribute_instance } modport_clocking_declaration

modport_clocking_declaration ::= **clocking** clocking_identifier

modport_simple_ports_declaration ::=
      port_direction modport_simple_port { **,** modport_simple_port }

modport_simple_port ::=
      port_identifier
    | **.** port_identifier **(** [expression] **)**

modport_tf_ports_declaration ::=
      import_export modport_tf_port { **,** modport_tf_port }

modport_tf_port ::=
      method_prototype
    | tf_identifier

import_export ::= **import** | **export**

A.2.10 Assertion declarations

concurrent_assertion_item ::= [block_identifier **:**] concurrent_assertion_statement

concurrent_assertion_statement ::=
      assert_property_statement
    | assume_property_statement
    | cover_property_statement

assert_property_statement::=
      assert property (property_spec **)** action_block

assume_property_statement::=
      assume property (property_spec **) ;**

cover_property_statement::=
      cover property (property_spec **)** statement_or_null

expect_property_statement ::=
      expect (property_spec **)** action_block

property_instance ::=
      ps_property_identifier [**(** [list_of_arguments] **)**]

concurrent_assertion_item_declaration ::=
      property_declaration
    | sequence_declaration

property_declaration ::=
      property property_identifier [**(** [tf_port_list] **)**] **;**
         { assertion_variable_declaration }
         property_spec **;**
      endproperty [**:** property_identifier]

property_spec ::=
      [clocking_event] [**disable iff (** expression_or_dist **)**] property_expr

property_expr ::=
      sequence_expr
    | **(** property_expr **)**
    | **not** property_expr
    | property_expr **or** property_expr
    | property_expr **and** property_expr
    | sequence_expr **|->** property_expr
    | sequence_expr **|=>** property_expr
    | **if (** expression_or_dist **)** property_expr [**else** property_expr]
    | property_instance
    | clocking_event property_expr

sequence_declaration ::=
    sequence sequence_identifier [**(** [tf_port_list] **)**] **;**
        { assertion_variable_declaration }
        sequence_expr **;**
    endsequence [**:** sequence_identifier]

sequence_expr ::=
    cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
    | sequence_expr cycle_delay_range sequence_expr { cycle_delay_range sequence_expr }
    | expression_or_dist [boolean_abbrev]
    | **(** expression_or_dist {**,** sequence_match_item } **)** [boolean_abbrev]
    | sequence_instance [sequence_abbrev]
    | **(** sequence_expr {**,** sequence_match_item } **)** [sequence_abbrev]
    | sequence_expr **and** sequence_expr
    | sequence_expr **intersect** sequence_expr
    | sequence_expr **or** sequence_expr
    | **first_match (** sequence_expr {**,** sequence_match_item} **)**
    | expression_or_dist **throughout** sequence_expr
    | sequence_expr **within** sequence_expr
    | clocking_event sequence_expr

cycle_delay_range ::=
    ## integral_number
    | **##** identifier
    | **## (** constant_expression **)**
    | **## [** cycle_delay_const_range_expression **]**

sequence_method_call ::=
    sequence_instance **.** method_identifier

sequence_match_item ::=
    operator_assignment
    | inc_or_dec_expression
    | subroutine_call

sequence_instance ::=
    ps_sequence_identifier [**(** [list_of_arguments] **)**]

formal_list_item ::=
    formal_identifier [**=** actual_arg_expr]

list_of_formals ::= formal_list_item { **,** formal_list_item }

actual_arg_expr ::=
    event_expression
    | **$**

boolean_abbrev ::=
    consecutive_repetition
    | non_consecutive_repetition
    | goto_repetition

sequence_abbrev ::= consecutive_repetition

consecutive_repetition ::= **[*** const_or_range_expression **]**

non_consecutive_repetition ::= **[=** const_or_range_expression **]**

goto_repetition ::= **[->** const_or_range_expression **]**

const_or_range_expression ::=
    constant_expression
    | cycle_delay_const_range_expression

cycle_delay_const_range_expression ::=

constant_expression **:** constant_expression
| constant_expression **: $**

expression_or_dist ::= expression [**dist {** dist_list **}**]

assertion_variable_declaration ::=
    var_data_type list_of_variable_identifiers **;**

A.2.11 Covergroup declarations

covergroup_declaration ::=
    covergroup covergroup_identifier [**(** [tf_port_list] **)**] [coverage_event] **;**
        { coverage_spec_or_option }
    endgroup [**:** covergroup_identifier]

coverage_spec_or_option ::=
    {attribute_instance} coverage_spec
    | {attribute_instance} coverage_option **;**

coverage_option ::=
    option.member_identifier **=** expression
    | **type_option.**member_identifier **=** expression

coverage_spec ::=
    cover_point
    | cover_cross

coverage_event ::=
    clocking_event
    | **@@(** block_event_expression **)**

block_event_expression ::=
    block_event_expression **or** block_event_expression
    | **begin** hierarchical_btf_identifier
    | **end** hierarchical_btf_identifier

hierarchical_btf_identifier ::=
    hierarchical_tf_identifier
    | hierarchical_block_identifier
    | hierarchical_identifier [class_scope] method_identifier

cover_point ::= [cover_point_identifier **:**] **coverpoint** expression [**iff (** expression **)**] bins_or_empty

bins_or_empty ::=
    { {attribute_instance} { bins_or_options **;** } **}**
    | **;**

bins_or_options ::=
    coverage_option
    | [**wildcard**] bins_keyword bin_identifier [**[** [expression] **]**] **= {** open_range_list **}** [**iff (** expression
)]
    | [**wildcard**] bins_keyword bin_identifier [**[ ]**] **=** trans_list [**iff (** expression **)**]
    | bins_keyword bin_identifier [**[** [expression] **]**] **= default** [**iff (** expression **)**]
    | bins_keyword bin_identifier **= default sequence** [**iff (** expression **)**]

bins_keyword::= **bins** | **illegal_bins** | **ignore_bins**

range_list ::= value_range { **,** value_range }

trans_list ::= **(** trans_set **)** { **, (** trans_set **)** }

trans_set ::= trans_range_list { **=>** trans_range_list }

trans_range_list ::=
    trans_item

    | trans_item [**[*** repeat_range **]**]
    | trans_item [**[–>** repeat_range **]**]
    | trans_item [**[=** repeat_range **]**]

trans_item ::= range_list

repeat_range ::=
    expression
    | expression **:** expression

cover_cross ::= [cover_point_identifier **:**] **cross** list_of_coverpoints [**iff (** expression **)**]
    select_bins_or_empty

list_of_coverpoints ::= cross_item **,** cross_item { **,** cross_item }

cross_item ::=
    cover_point_identifier
    | variable_identifier

select_bins_or_empty ::=
    { { bins_selection_or_option **;** } **}**
    | **;**

bins_selection_or_option ::=
    { attribute_instance } coverage_option
    | { attribute_instance } bins_selection

bins_selection ::= bins_keyword bin_identifier **=** select_expression [**iff (** expression **)**]

select_expression ::=
    select_condition
    | **!** select_condition
    | select_expression **&&** select_expression
    | select_expression **||** select_expression
    | **(** select_expression **)**

select_condition ::= **binsof (** bins_expression **)** [**intersect {** open_range_list **}**]

bins_expression ::=
    variable_identifier
    | cover_point_identifier [**.** bins_identifier]

open_range_list ::= open_value_range { **,** open_value_range }

open_value_range ::= value_range[20]

A.3 Primitive instances

A.3.1 Primitive instantiation and instances

gate_instantiation ::=
    cmos_switchtype [delay3] cmos_switch_instance { **,** cmos_switch_instance } **;**
    | enable_gatetype [drive_strength] [delay3] enable_gate_instance { **,** enable_gate_instance } **;**
    | mos_switchtype [delay3] mos_switch_instance { **,** mos_switch_instance } **;**
    | n_input_gatetype [drive_strength] [delay2] n_input_gate_instance { **,** n_input_gate_instance } **;**
    | n_output_gatetype [drive_strength] [delay2] n_output_gate_instance
        { **,** n_output_gate_instance } **;**
    | pass_en_switchtype [delay2] pass_enable_switch_instance { , pass_enable_switch_instance } **;**
    | pass_switchtype pass_switch_instance { **,** pass_switch_instance } **;**
    | **pulldown** [pulldown_strength] pull_gate_instance { **,** pull_gate_instance } **;**
    | **pullup** [pullup_strength] pull_gate_instance { **,** pull_gate_instance } **;**

cmos_switch_instance ::= [name_of_instance] **(** output_terminal **,** input_terminal **,**
         ncontrol_terminal **,** pcontrol_terminal **)**

enable_gate_instance ::= [name_of_instance] **(** output_terminal **,** input_terminal **,** enable_terminal **)**

mos_switch_instance ::= [name_of_instance] **(** output_terminal **,** input_terminal **,** enable_terminal **)**

n_input_gate_instance ::= [name_of_instance] **(** output_terminal **,** input_terminal { **,** input_terminal } **)**

n_output_gate_instance ::= [name_of_instance] **(** output_terminal { **,** output_terminal } **,**
         input_terminal **)**

pass_switch_instance ::= [name_of_instance] **(** inout_terminal **,** inout_terminal **)**

pass_enable_switch_instance ::= [name_of_instance] **(** inout_terminal , inout_terminal **,**
         enable_terminal **)**

pull_gate_instance ::= [name_of_instance] **(** output_terminal **)**

A.3.2 Primitive strengths

pulldown_strength ::=
         (strength0 **,** strength1 **)**
       | **(** strength1 **,** strength0 **)**
       | **(** strength0 **)**

pullup_strength ::=
         (strength0 **,** strength1 **)**
       | **(** strength1 **,** strength0 **)**
       | **(** strength1 **)**

A.3.3 Primitive terminals

enable_terminal ::= expression

inout_terminal ::= net_lvalue

input_terminal ::= expression

ncontrol_terminal ::= expression

output_terminal ::= net_lvalue

pcontrol_terminal ::= expression

A.3.4 Primitive gate and switch types

cmos_switchtype ::= **cmos** | **rcmos**

enable_gatetype ::= **bufif0** | **bufif1** | **notif0** | **notif1**

mos_switchtype ::= **nmos** | **pmos** | **rnmos** | **rpmos**

n_input_gatetype ::= **and** | **nand** | **or** | **nor** | **xor** | **xnor**

n_output_gatetype ::= **buf** | **not**

pass_en_switchtype ::= **tranif0** | **tranif1** | **rtranif1** | **rtranif0**

pass_switchtype ::= **tran** | **rtran**

A.4 Module, interface and generated instantiation

A.4.1 Instantiation

A.4.1.1 Module instantiation

module_instantiation ::=
      module_identifier [parameter_value_assignment] hierarchical_instance { **,** hierarchical_instance } **;**

parameter_value_assignment ::= **# (** [list_of_parameter_assignments] **)**

list_of_parameter_assignments ::=
      ordered_parameter_assignment { **,** ordered_parameter_assignment }
    | named_parameter_assignment { **,** named_parameter_assignment }

ordered_parameter_assignment ::= param_expression

named_parameter_assignment ::= **.** parameter_identifier **(** [param_expression] **)**

hierarchical_instance ::= name_of_instance **(** [list_of_port_connections] **)**

name_of_instance ::= instance_identifier { unpacked_dimension }

list_of_port_connections[16] ::=
      ordered_port_connection { **,** ordered_port_connection }
    | named_port_connection { **,** named_port_connection }

ordered_port_connection ::= { attribute_instance } [expression]

named_port_connection ::=
      { attribute_instance } **.** port_identifier [**(** [expression] **)**]
    | { attribute_instance } **.***

A.4.1.2 Interface instantiation

interface_instantiation ::=
      interface_identifier [parameter_value_assignment] hierarchical_instance { **,** hierarchical_instance }
      ;

A.4.1.3 Program instantiation

program_instantiation ::=
      program_identifier [parameter_value_assignment] hierarchical_instance { **,** hierarchical_instance }
      ;

A.4.2 Generated instantiation

module_or_interface_or_generate_item[31] ::=
      module_or_generate_item
    | interface_or_generate_item

generate_region ::=
      generate { module_or_interface_or_generate_item } **endgenerate**

loop_generate_construct ::=
      for (genvar_initialization **;** genvar_expression **;** genvar_iteration **)**
        generate_block

genvar_initialization ::=
      [**genvar**] genvar_identifier **=** constant_expression

genvar_iteration ::=
        genvar_identifier assignment_operator genvar_expression
     | inc_or_dec_operator genvar_identifier
     | genvar_identifier inc_or_dec_operator

conditional_generate_construct ::=
        if_generate_construct
     | case_generate_construct

if_generate_construct ::=
        if (constant_expression **)** generate_block_or_null [**else** generate_block_or_null]

case_generate_construct ::=
        case (constant_expression **)** case_generate_item { case_generate_item } **endcase**

case_generate_item ::=
        constant_expression { **,** constant_expression } **:** generate_block_or_null
     | **default** [**:**] generate_block_or_null

generate_block ::=
        module_or_interface_or_generate_item
     | [generate_block_identifier **:**] **begin** [**:** generate_block_identifier]
          { module_or_interface_or_generate_item }
      end [**:** generate_block_identifier]

generate_block_or_null ::= generate_block | **;**

A.5 UDP declaration and instantiation

A.5.1 UDP declaration

udp_nonansi_declaration ::=
        { attribute_instance } **primitive** udp_identifier **(** udp_port_list **) ;**

udp_ansi_declaration ::=
        { attribute_instance } **primitive** udp_identifier **(** udp_declaration_port_list **) ;**

udp_declaration ::=
        udp_nonansi_declaration udp_port_declaration { udp_port_declaration }
          udp_body
      endprimitive [**:** udp_identifier]
     | udp_ansi_declaration
          udp_body
      endprimitive [**:** udp_identifier]
     | **extern** udp_nonansi_declaration
     | **extern** udp_ansi_declaration
     | { attribute_instance } **primitive** udp_identifier **(.*) ;**
          { udp_port_declaration }
          udp_body
      endprimitive [**:** udp_identifier]

A.5.2 UDP ports

udp_port_list ::= output_port_identifier **,** input_port_identifier { **,** input_port_identifier }

udp_declaration_port_list ::= udp_output_declaration **,** udp_input_declaration { **,** udp_input_declaration }

udp_port_declaration ::=
        udp_output_declaration **;**

    | udp_input_declaration **;**
    | udp_reg_declaration **;**

udp_output_declaration ::=
    { attribute_instance } **output** port_identifier
    | { attribute_instance } **output reg** port_identifier [**=** constant_expression]

udp_input_declaration ::= { attribute_instance } **input** list_of_udp_port_identifiers

udp_reg_declaration ::= { attribute_instance } **reg** variable_identifier

A.5.3 UDP body

udp_body ::= combinational_body | sequential_body

combinational_body ::= **table** combinational_entry { combinational_entry } **endtable**

combinational_entry ::= level_input_list **:** output_symbol **;**

sequential_body ::= [udp_initial_statement] **table** sequential_entry { sequential_entry } **endtable**

udp_initial_statement ::= **initial** output_port_identifier **=** init_val **;**

init_val ::= **1'b0** | **1'b1** | **1'bx** | **1'bX** | **1'B0** | **1'B1** | **1'Bx** | **1'BX** | **1** | **0**

sequential_entry ::= seq_input_list **:** current_state **:** next_state **;**

seq_input_list ::= level_input_list | edge_input_list

level_input_list ::= level_symbol { level_symbol }

edge_input_list ::= { level_symbol } edge_indicator { level_symbol }

edge_indicator ::= (level_symbol level_symbol) | edge_symbol

current_state ::= level_symbol

next_state ::= output_symbol | **-**

output_symbol ::= **0** | **1** | **x** | **X**

level_symbol ::= **0** | **1** | **x** | **X** | **?** | **b** | **B**

edge_symbol ::= **r** | **R** | **f** | **F** | **p** | **P** | **n** | **N** | *

A.5.4 UDP instantiation

udp_instantiation ::= udp_identifier [drive_strength] [delay2] udp_instance { **,** udp_instance } **;**

udp_instance ::= [name_of_instance] **(** output_terminal **,** input_terminal { **,** input_terminal } **)**

A.6 Behavioral statements

A.6.1 Continuous assignment and net alias statements

continuous_assign ::=
    assign [drive_strength] [delay3] list_of_net_assignments **;**
    | **assign** [delay_control] list_of_variable_assignments **;**

list_of_net_assignments ::= net_assignment { **,** net_assignment }

list_of_variable_assignments ::= variable_assignment { **,** variable_assignment }

net_alias ::= **alias** net_lvalue **=** net_lvalue { **=** net_lvalue } **;**

net_assignment ::= net_lvalue **=** expression

A.6.2 Procedural blocks and assignments

initial_construct ::= **initial** statement_or_null

always_construct ::= always_keyword statement

always_keyword ::= **always** | **always_comb** | **always_latch** | **always_ff**

final_construct ::= **final** function_statement

blocking_assignment ::=
      variable_lvalue **=** delay_or_event_control expression
    | hierarchical_dynamic_array_variable_identifier **=** dynamic_array_new
    | [implicit_class_handle **.** | class_scope | package_scope] hierarchical_variable_identifier
        select **=** class_new
    | operator_assignment

operator_assignment ::= variable_lvalue assignment_operator expression

assignment_operator ::=
      = | **+=** | **-=** | ***=** | **/=** | **%=** | **&=** | **|=** | **^=** | **<<=** | **>>=** | **<<<=** | **>>>=**

nonblocking_assignment ::= variable_lvalue **<=** [delay_or_event_control] expression

procedural_continuous_assignment ::=
      assign variable_assignment
    | **deassign** variable_lvalue
    | **force** variable_assignment
    | **force** net_assignment
    | **release** variable_lvalue
    | **release** net_lvalue

variable_assignment ::= variable_lvalue **=** expression

A.6.3 Parallel and sequential blocks

action_block ::=
      statement_or_null
    | [statement] **else** statement_or_null

seq_block ::=
      begin [**:** block_identifier] { block_item_declaration } { statement_or_null }
      end [**:** block_identifier]

par_block ::=
      fork [**:** block_identifier] { block_item_declaration } { statement_or_null }
      join_keyword [**:** block_identifier]

join_keyword ::= **join** | **join_any** | **join_none**

A.6.4 Statements

statement_or_null ::=
      statement
    | { attribute_instance } **;**

statement ::= [block_identifier **:**] { attribute_instance } statement_item

statement_item ::=
      blocking_assignment **;**
    | nonblocking_assignment **;**
    | procedural_continuous_assignment **;**
    | case_statement

    | conditional_statement
    | inc_or_dec_expression **;**
    | subroutine_call_statement
    | disable_statement
    | event_trigger
    | loop_statement
    | jump_statement
    | par_block
    | procedural_timing_control_statement
    | seq_block
    | wait_statement
    | procedural_assertion_statement
    | clocking_drive **;**
    | randsequence_statement
    | randcase_statement
    | expect_property_statement

function_statement ::= statement

function_statement_or_null ::=
    function_statement
    | { attribute_instance } **;**

variable_identifier_list ::= variable_identifier { **,** variable_identifier }

A.6.5 Timing control statements

procedural_timing_control_statement ::=
    procedural_timing_control statement_or_null

delay_or_event_control ::=
    delay_control
    | event_control
    | **repeat (** expression **)** event_control

delay_control ::=
    # delay_value
    | **# (** mintypmax_expression **)**

event_control ::=
    @ hierarchical_event_identifier
    | **@ (** event_expression **)**
    | **@***
    | **@ (*)**
    | **@** sequence_instance

event_expression ::=
    [edge_identifier] expression [**iff** expression]
    | sequence_instance [**iff** expression]
    | event_expression **or** event_expression
    | event_expression **,** event_expression

procedural_timing_control ::=
    delay_control
    | event_control
    | cycle_delay

jump_statement ::=
    return [expression] **;**
    | **break ;**

    | **continue ;**

wait_statement ::=
    wait (expression **)** statement_or_null
    | **wait fork ;**
    | **wait_order (** hierarchical_identifier { **,** hierarchical_identifier } **)** action_block

event_trigger ::=
    -> hierarchical_event_identifier **;**
    |**>>** [delay_or_event_control] hierarchical_event_identifier **;**

disable_statement ::=
    disable hierarchical_task_identifier **;**
    | **disable** hierarchical_block_identifier **;**
    | **disable fork ;**

A.6.6 Conditional statements

conditional_statement ::=
    if (cond_predicate **)** statement_or_null [**else** statement_or_null]
    | unique_priority_if_statement

unique_priority_if_statement ::=
    [unique_priority] **if (** cond_predicate **)** statement_or_null
      { **else if (** cond_predicate **)** statement_or_null }
      [**else** statement_or_null]

unique_priority ::= **unique** | **priority**

cond_predicate ::=
    expression_or_cond_pattern { **&&&** expression_or_cond_pattern }

expression_or_cond_pattern ::=
    expression | cond_pattern

cond_pattern ::= expression **matches** pattern

A.6.7 `case` statements

case_statement ::=
    [unique_priority] case_keyword **(** expression **)** case_item { case_item } **endcase**
    | [unique_priority] case_keyword **(** expression **) matches** case_pattern_item { case_pattern_item }
      endcase
    | [unique_priority] **case (** expression **) inside** case_inside_item { case_inside_item } **endcase**

case_keyword ::= **case** | **casez** | **casex**

case_item ::=
    expression { **,** expression } **:** statement_or_null
    | **default** [**:**] statement_or_null

case_pattern_item ::=
    pattern [**&&&** expression] **:** statement_or_null
    | **default** [**:**] statement_or_null

case_inside_item ::=
    open_range_list **:** statement_or_null
    | **default** [**:**] statement_or_null

randcase_statement ::=
    randcase randcase_item { randcase_item } **endcase**

randcase_item ::= expression **:** statement_or_null

A.6.7.1 Patterns

pattern ::=
    . variable_identifier
    | **.***
    | constant_expression
    | **tagged** member_identifier [pattern]
    | **'{** pattern { **,** pattern } **}**
    | **'{** member_identifier **:** pattern { **,** member_identifier **:** pattern } **}**

assignment_pattern ::=
    '{ expression { **,** expression } **}**
    | **'{** structure_pattern_key **:** expression { **,** structure_pattern_key **:** expression } **}**
    | **'{** array_pattern_key **:** expression { **,** array_pattern_key **:** expression } **}**
    | **'{** constant_expression **{** expression { **,** expression } **} }**

structure_pattern_key ::= member_identifier | assignment_pattern_key

array_pattern_key ::= constant_expression | assignment_pattern_key

assignment_pattern_key ::= simple_type | **default**

assignment_pattern_expression ::=
    [assignment_pattern_expression_type] assignment_pattern

assignment_pattern_expression_type ::= ps_type_identifier | ps_parameter_identifier | integer_atom_type

constant_assignment_pattern_expression[35] ::= assignment_pattern_expression

assignment_pattern_net_lvalue ::=
    '{ net_lvalue {**,** net_lvalue } **}**

assignment_pattern_variable_lvalue ::=
    '{ variable_lvalue {**,** variable_lvalue } **}**

A.6.8 Looping statements

loop_statement ::=
    forever statement_or_null
    | **repeat (** expression **)** statement_or_null
    | **while (** expression **)** statement_or_null
    | **for (** for_initialization **;** expression **;** for_step **)**
        statement_or_null
    | **do** statement_or_null **while (** expression **) ;**
    | **foreach (** array_identifier **[** loop_variables **])** statement

for_initialization ::=
    list_of_variable_assignments
    | for_variable_declaration { **,** for_variable_declaration }

for_variable_declaration ::=
    data_type variable_identifier **=** expression { **,** variable_identifier **=** expression }

for_step ::= for_step_assignment { **,** for_step_assignment }

for_step_assignment ::=
    operator_assignment
    | inc_or_dec_expression
    | function_subroutine_call

loop_variables ::= [index_variable_identifier] { **,** [index_variable_identifier] }

A.6.9 Subroutine call statements

subroutine_call_statement ::=
        subroutine_call **;**
    | **void ' (** function_subroutine_call **) ;**

A.6.10 Assertion statements

procedural_assertion_statement ::=
        concurrent_assertion_statement
    | immediate_assert_statement

immediate_assert_statement ::=
        assert (expression **)** action_block

A.6.11 Clocking block

clocking_declaration ::= [**default**] **clocking** [clocking_identifier] clocking_event **;**
        { clocking_item }
        endclocking [**:** clocking_identifier]

clocking_event ::=
    @ identifier
    | **@ (** event_expression **)**

clocking_item ::=
        default default_skew **;**
    | clocking_direction list_of_clocking_decl_assign **;**
    | { attribute_instance } concurrent_assertion_item_declaration

default_skew ::=
        input clocking_skew
    | **output** clocking_skew
    | **input** clocking_skew **output** clocking_skew

clocking_direction ::=
        input [clocking_skew]
    | **output** [clocking_skew]
    | **input** [clocking_skew] **output** [clocking_skew]
    | **inout**

list_of_clocking_decl_assign ::= clocking_decl_assign { , clocking_decl_assign }

clocking_decl_assign ::= signal_identifier [**=** expression]

clocking_skew ::=
        edge_identifier [delay_control]
    | delay_control

clocking_drive ::=
        clockvar_expression **<=** [cycle_delay] expression
    | cycle_delay clockvar_expression **<=** expression

cycle_delay ::=
    ## integral_number
    | **##** identifier
    | **## (** expression **)**

clockvar ::= hierarchical_identifier

clockvar_expression ::= clockvar select

A.6.12 Randsequence

randsequence_statement ::= **randsequence (** [production_identifier] **)**
        production { production }
    endsequence

production ::= [function_data_type] production_identifier [**(** tf_port_list **)**] **:** rs_rule { **|** rs_rule } **;**

rs_rule ::= rs_production_list [**:=** weight_specification [rs_code_block]]

rs_production_list ::=
        rs_prod { rs_prod }
    | **rand join** [**(** expression **)**] production_item production_item { production_item }

weight_specification ::=
        integral_number
    | ps_identifier
    | **(** expression **)**

rs_code_block ::= **{** { data_declaration } { statement_or_null } **}**

rs_prod ::=
        production_item
    | rs_code_block
    | rs_if_else
    | rs_repeat
    | rs_case

production_item ::= production_identifier [**(** list_of_arguments **)**]

rs_if_else ::= **if (** expression **)** production_item [**else** production_item]

rs_repeat ::= **repeat (** expression **)** production_item

rs_case ::= **case (** expression **)** rs_case_item { rs_case_item } **endcase**

rs_case_item ::=
        expression { **,** expression } **:** production_item **;**
    | **default** [**:**] production_item **;**

A.7 Specify section

A.7.1 Specify block declaration

specify_block ::= **specify** { specify_item } **endspecify**

specify_item ::=
        specparam_declaration
    | pulsestyle_declaration
    | showcancelled_declaration
    | path_declaration
    | system_timing_check

pulsestyle_declaration ::=
        pulsestyle_onevent list_of_path_outputs **;**
    | **pulsestyle_ondetect** list_of_path_outputs **;**

showcancelled_declaration ::=
        showcancelled list_of_path_outputs **;**
    | **noshowcancelled** list_of_path_outputs **;**

A.7.2 Specify path declarations

path_declaration ::=
      simple_path_declaration **;**
    | edge_sensitive_path_declaration **;**
    | state_dependent_path_declaration **;**

simple_path_declaration ::=
      parallel_path_description **=** path_delay_value
    | full_path_description **=** path_delay_value

parallel_path_description ::=
      (specify_input_terminal_descriptor [polarity_operator] **=>** specify_output_terminal_descriptor **)**

full_path_description ::=
      (list_of_path_inputs [polarity_operator] ***>** list_of_path_outputs **)**

list_of_path_inputs ::=
      specify_input_terminal_descriptor { **,** specify_input_terminal_descriptor }

list_of_path_outputs ::=
      specify_output_terminal_descriptor { **,** specify_output_terminal_descriptor }

A.7.3 Specify block terminals

specify_input_terminal_descriptor ::=
      input_identifier [**[** constant_range_expression **]**]

specify_output_terminal_descriptor ::=
      output_identifier [**[** constant_range_expression **]**]

input_identifier ::= input_port_identifier | inout_port_identifier | interface_identifier.port_identifier

output_identifier ::= output_port_identifier | inout_port_identifier | interface_identifier.port_identifier

A.7.4 Specify path delays

path_delay_value ::=
      list_of_path_delay_expressions
    | **(** list_of_path_delay_expressions **)**

list_of_path_delay_expressions ::=
      t_path_delay_expression
    | trise_path_delay_expression **,** tfall_path_delay_expression
    | trise_path_delay_expression **,** tfall_path_delay_expression **,** tz_path_delay_expression
    | t01_path_delay_expression **,** t10_path_delay_expression **,** t0z_path_delay_expression **,**
        tz1_path_delay_expression **,** t1z_path_delay_expression **,** tz0_path_delay_expression
    | t01_path_delay_expression **,** t10_path_delay_expression **,** t0z_path_delay_expression **,**
        tz1_path_delay_expression **,** t1z_path_delay_expression **,** tz0_path_delay_expression **,**
        t0x_path_delay_expression **,** tx1_path_delay_expression **,** t1x_path_delay_expression **,**
        tx0_path_delay_expression **,** txz_path_delay_expression **,** tzx_path_delay_expression

t_path_delay_expression ::= path_delay_expression

trise_path_delay_expression ::= path_delay_expression

tfall_path_delay_expression ::= path_delay_expression

tz_path_delay_expression ::= path_delay_expression

t01_path_delay_expression ::= path_delay_expression

t10_path_delay_expression ::= path_delay_expression

t0z_path_delay_expression ::= path_delay_expression

tz1_path_delay_expression ::= path_delay_expression

t1z_path_delay_expression ::= path_delay_expression

tz0_path_delay_expression ::= path_delay_expression

t0x_path_delay_expression ::= path_delay_expression

tx1_path_delay_expression ::= path_delay_expression

t1x_path_delay_expression ::= path_delay_expression

tx0_path_delay_expression ::= path_delay_expression

txz_path_delay_expression ::= path_delay_expression

tzx_path_delay_expression ::= path_delay_expression

path_delay_expression ::= constant_mintypmax_expression

edge_sensitive_path_declaration ::=
     parallel_edge_sensitive_path_description **=** path_delay_value
   | full_edge_sensitive_path_description **=** path_delay_value

parallel_edge_sensitive_path_description ::=
    ([edge_identifier] specify_input_terminal_descriptor **=>**
      (specify_output_terminal_descriptor [polarity_operator] **:** data_source_expression **))**

full_edge_sensitive_path_description ::=
    ([edge_identifier] list_of_path_inputs ***>**
      (list_of_path_outputs [polarity_operator] **:** data_source_expression **))**

data_source_expression ::= expression

edge_identifier ::= **posedge** | **negedge**

state_dependent_path_declaration ::=
    if (module_path_expression **)** simple_path_declaration
   | **if (** module_path_expression **)** edge_sensitive_path_declaration
   | **ifnone** simple_path_declaration

polarity_operator ::= **+** | **-**

A.7.5 System timing checks

A.7.5.1 System timing check commands

system_timing_check ::=
    $setup_timing_check
   | $hold_timing_check
   | $setuphold_timing_check
   | $recovery_timing_check
   | $removal_timing_check
   | $recrem_timing_check
   | $skew_timing_check
   | $timeskew_timing_check
   | $fullskew_timing_check
   | $period_timing_check
   | $width_timing_check
   | $nochange_timing_check

$setup_timing_check ::=
    $setup (data_event **,** reference_event **,** timing_check_limit [**,** [notifier]] **) ;**

$hold_timing_check ::=

$hold (reference_event , data_event , timing_check_limit [, [notifier]]) ;
$setuphold_timing_check ::=
    $setuphold (reference_event , data_event , timing_check_limit , timing_check_limit
      [, [notifier] [, [stamptime_condition] [, [checktime_condition]
      [, [delayed_reference] [, [delayed_data]]]]]]) ;
$recovery_timing_check ::=
    $recovery (reference_event , data_event , timing_check_limit [, [notifier]]) ;
$removal_timing_check ::=
    $removal (reference_event , data_event , timing_check_limit [, [notifier]]) ;
$recrem_timing_check ::=
    $recrem (reference_event , data_event , timing_check_limit , timing_check_limit
      [, [notifier] [, [stamptime_condition] [, [checktime_condition]
      [, [delayed_reference] [, [delayed_data]]]]]]) ;
$skew_timing_check ::=
    $skew (reference_event , data_event , timing_check_limit [, [notifier]]) ;
$timeskew_timing_check ::=
    $timeskew (reference_event , data_event , timing_check_limit
      [, [notifier] [, [event_based_flag] [, [remain_active_flag]]]]) ;
$fullskew_timing_check ::=
    $fullskew (reference_event , data_event , timing_check_limit , timing_check_limit
      [, [notifier] [, [event_based_flag] [, [remain_active_flag]]]]) ;
$period_timing_check ::=
    $period (controlled_reference_event , timing_check_limit [, [notifier]]) ;
$width_timing_check ::=
    $width (controlled_reference_event , timing_check_limit , threshold [, [notifier]]) ;
$nochange_timing_check ::=
    $nochange (reference_event , data_event , start_edge_offset ,
      end_edge_offset [, [notifier]]) ;

A.7.5.2 System timing check command arguments

checktime_condition ::= mintypmax_expression

controlled_reference_event ::= controlled_timing_check_event

data_event ::= timing_check_event

delayed_data ::=
    terminal_identifier
  | terminal_identifier [constant_mintypmax_expression]

delayed_reference ::=
    terminal_identifier
  | terminal_identifier [constant_mintypmax_expression]

end_edge_offset ::= mintypmax_expression

event_based_flag ::= constant_expression

notifier ::= variable_identifier

reference_event ::= timing_check_event

remain_active_flag ::= constant_mintypmax_expression

stamptime_condition ::= mintypmax_expression

start_edge_offset ::= mintypmax_expression

threshold ::=constant_expression

timing_check_limit ::= expression

A.7.5.3 System timing check event definitions

timing_check_event ::=
      [timing_check_event_control] specify_terminal_descriptor [**&&&** timing_check_condition]
controlled_timing_check_event ::=
      timing_check_event_control specify_terminal_descriptor [**&&&** timing_check_condition]
timing_check_event_control ::=
      posedge
      | **negedge**
      | edge_control_specifier
specify_terminal_descriptor ::=
      specify_input_terminal_descriptor
      | specify_output_terminal_descriptor
edge_control_specifier ::= **edge [** edge_descriptor { **,** edge_descriptor } **]**

edge_descriptor[1] ::= **01** | **10** | z_or_x zero_or_one | zero_or_one z_or_x

zero_or_one ::= **0** | **1**

z_or_x ::= **x** | **X** | **z** | **Z**

timing_check_condition ::=
      scalar_timing_check_condition
      | **(** scalar_timing_check_condition **)**
scalar_timing_check_condition ::=
      expression
      | **~** expression
      | expression **==** scalar_constant
      | expression **===** scalar_constant
      | expression **!=** scalar_constant
      | expression **!==** scalar_constant
scalar_constant ::= **1'b0** | **1'b1** | **1'B0** | **1'B1** | **'b0** | **'b1** | **'B0** | **'B1** | **1** | **0**

A.8 Expressions

A.8.1 Concatenations

concatenation ::=
      { expression { **,** expression } **}**
constant_concatenation ::=
      { constant_expression { **,** constant_expression } **}**
constant_multiple_concatenation ::= **{** constant_expression constant_concatenation **}**
module_path_concatenation ::= **{** module_path_expression { **,** module_path_expression } **}**
module_path_multiple_concatenation ::= **{** constant_expression module_path_concatenation **}**
multiple_concatenation ::= **{** expression concatenation **}**[18]
streaming_concatenation ::= **{** stream_operator [slice_size] stream_concatenation **}**
stream_operator ::= **>>** | **<<**
slice_size ::= simple_type | constant_expression
stream_concatenation ::= **{** stream_expression { **,** stream_expression } **}**

stream_expression ::= expression [**with [** array_range_expression **]**]

array_range_expression ::=
      expression
     | expression **:** expression
     | expression **+:** expression
     | expression **-:** expression

empty_queue[21] ::= **{ }**

A.8.2 Subroutine calls

constant_function_call ::= function_subroutine_call[24]

tf_call[36] ::= ps_or_hierarchical_tf_identifier { attribute_instance } [**(** list_of_arguments **)**]

system_tf_call ::=
      system_tf_identifier [**(** list_of_arguments **)**]
     | system_tf_identifier **(** data_type [**,** expression] **)**

subroutine_call ::=
      tf_call
     | system_tf_call
     | method_call
     | randomize_call

function_subroutine_call ::= subroutine_call

list_of_arguments ::=
      [expression] { **,** [expression] } { **, .** identifier **(** [expression] **)** }
     | **.** identifier **(** [expression] **)** { **, .** identifier **(** [expression] **)** }

method_call ::= method_call_root **.** method_call_body

method_call_body ::=
      method_identifier { attribute_instance } [**(** list_of_arguments **)**]
     | built_in_method_call

built_in_method_call ::=
      array_manipulation_call
     | randomize_call

array_manipulation_call ::=
      array_method_name { attribute_instance }
        [**(** list_of_arguments **)**]
        [**with (** expression **)**]

randomize_call ::=
      randomize { attribute_instance }
        [**(** [variable_identifier_list | **null**] **)**]
        [**with** constraint_block]

method_call_root ::= expression | implicit_class_handle

array_method_name ::=
      method_identifier | **unique** | **and** | **or** | **xor**

A.8.3 Expressions

inc_or_dec_expression ::=
      inc_or_dec_operator { attribute_instance } variable_lvalue
     | variable_lvalue { attribute_instance } inc_or_dec_operator

conditional_expression ::= cond_predicate **?** { attribute_instance } expression **:** expression

constant_expression ::=
      constant_primary
    | unary_operator { attribute_instance } constant_primary
    | constant_expression binary_operator { attribute_instance } constant_expression
    | constant_expression **?** { attribute_instance } constant_expression **:** constant_expression

constant_mintypmax_expression ::=
      constant_expression
    | constant_expression **:** constant_expression **:** constant_expression

constant_param_expression ::=
      constant_mintypmax_expression | data_type | **$**

param_expression ::= mintypmax_expression | data_type

constant_range_expression ::=
      constant_expression
    | constant_part_select_range

constant_part_select_range ::=
      constant_range
    | constant_indexed_range

constant_range ::= constant_expression **:** constant_expression

constant_indexed_range ::=
      constant_expression **+:** constant_expression
    | constant_expression **-:** constant_expression

expression ::=
      primary
    | unary_operator { attribute_instance } primary
    | inc_or_dec_expression
    | **(** operator_assignment **)**
    | expression binary_operator { attribute_instance } expression
    | conditional_expression
    | inside_expression
    | tagged_union_expression

tagged_union_expression ::=
      tagged member_identifier [expression]

inside_expression ::= expression **inside {** open_range_list **}**

value_range ::=
      expression
    | **[** expression **:** expression **]**

mintypmax_expression ::=
      expression
    | expression **:** expression **:** expression

module_path_conditional_expression ::= module_path_expression **?** { attribute_instance }
      module_path_expression **:** module_path_expression

module_path_expression ::=
      module_path_primary
    | unary_module_path_operator { attribute_instance } module_path_primary
    | module_path_expression binary_module_path_operator { attribute_instance }
      module_path_expression
    | module_path_conditional_expression

module_path_mintypmax_expression ::=
      module_path_expression

| module_path_expression **:** module_path_expression **:** module_path_expression

part_select_range ::= constant_range | indexed_range

indexed_range ::=
      expression **+:** constant_expression
    | expression **-:** constant_expression

genvar_expression ::= constant_expression

A.8.4 Primaries

constant_primary ::=
      primary_literal
    | ps_parameter_identifier constant_select
    | specparam_identifier [constant_range_expression]
    | genvar_identifier[32]
    | [package_scope | class_scope] enum_identifier
    | constant_concatenation
    | constant_multiple_concatenation
    | constant_function_call
    | (constant_mintypmax_expression)
    | constant_cast
    | constant_assignment_pattern_expression
    | type_reference[29]

module_path_primary ::=
      number
    | identifier
    | module_path_concatenation
    | module_path_multiple_concatenation
    | function_subroutine_call
    | **(** module_path_mintypmax_expression **)**

primary ::=
      primary_literal
    | [implicit_class_handle **.** | class_scope | package_scope] hierarchical_identifier select
    | empty_queue
    | concatenation
    | multiple_concatenation
    | function_subroutine_call
    | **(** mintypmax_expression **)**
    | cast
    | assignment_pattern_expression
    | streaming_concatenation
    | sequence_method_call
    | **this**[6]
    | **$**[22]
    | **null**

time_literal[5] ::=
      unsigned_number time_unit
    | fixed_point_number time_unit

time_unit ::= **s** | **ms** | **us** | **ns** | **ps** | **fs** | **step**

implicit_class_handle[6] ::= **this** | **super** | **this . super**

bit_select ::= { **[** expression **]** }

select ::=
     [{ **.** member_identifier bit_select } **.** member_identifier] bit_select [**[** part_select_range **]**]

constant_bit_select ::= { **[** constant_expression **]** }

constant_select ::=
     [{ **.** member_identifier constant_bit_select } **.** member_identifier] constant_bit_select
       [**[** constant_part_select_range **]**]

primary_literal ::= number | time_literal | unbased_unsized_literal | string_literal

constant_cast ::=
     casting_type **' (** constant_expression **)**

cast ::=
     casting_type **' (** expression **)**

A.8.5 Expression left-side values

net_lvalue ::=
     ps_or_hierarchical_net_identifier constant_select
     | **{** net_lvalue { **,** net_lvalue } **}**
     | [assignment_pattern_expression_type] assignment_pattern_net_lvalue

variable_lvalue ::=
     [implicit_class_handle **.** | package_scope] hierarchical_variable_identifier select
     | **{** variable_lvalue { **,** variable_lvalue } **}**
     | [assignment_pattern_expression_type] assignment_pattern_variable_lvalue
     | streaming_concatenation[30]

A.8.6 Operators

unary_operator ::=
     + | **-** | **!** | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^** | **^~**

binary_operator ::=
     + | **-** | ***** | **/** | **%** | **==** | **!=** | **===** | **!==** | **==?** | **!=?** | **&&** | **||** | ******
     | **<** | **<=** | **>** | **>=** | **&** | **|** | **^** | **^~** | **~^** | **>>** | **<<** | **>>>** | **<<<**

inc_or_dec_operator ::= **++** | **--**

unary_module_path_operator ::=
     ! | **~** | **&** | **~&** | **|** | **~|** | **^** | **~^** | **^~**

binary_module_path_operator ::=
     == | **!=** | **&&** | **||** | **&** | **|** | **^** | **^~** | **~^**

A.8.7 Numbers

number ::=
     integral_number
     | real_number

integral_number ::=
     decimal_number
     | octal_number
     | binary_number
     | hex_number

decimal_number ::=

            unsigned_number
        | [size] decimal_base unsigned_number
        | [size] decimal_base x_digit { _ }
        | [size] decimal_base z_digit { _ }

binary_number ::= [size] binary_base binary_value

octal_number ::= [size] octal_base octal_value

hex_number ::= [size] hex_base hex_value

sign ::= **+** | **-**

size ::= non_zero_unsigned_number

non_zero_unsigned_number[1] ::= non_zero_decimal_digit { _ | decimal_digit}

real_number[1] ::=
        fixed_point_number
        | unsigned_number [**.** unsigned_number] exp [sign] unsigned_number

fixed_point_number[1] ::= unsigned_number **.** unsigned_number

exp ::= **e** | **E**

unsigned_number[1] ::= decimal_digit { _ | decimal_digit }

binary_value[1] ::= binary_digit { _ | binary_digit }

octal_value[1] ::= octal_digit { _ | octal_digit }

hex_value[1] ::= hex_digit { _ | hex_digit }

decimal_base[1] ::= **'[s|S]d** | **'[s|S]D**

binary_base[1] ::= **'[s|S]b** | **'[s|S]B**

octal_base[1] ::= **'[s|S]o** | **'[s|S]O**

hex_base[1] ::= **'[s|S]h** | **'[s|S]H**

non_zero_decimal_digit ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

decimal_digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

binary_digit ::= x_digit | z_digit | **0** | **1**

octal_digit ::= x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

hex_digit ::= x_digit | z_digit | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **a** | **b** | **c** | **d** | **e** | **f** | **A** | **B** | **C** | **D** | **E** | **F**

x_digit ::= **x** | **X**

z_digit ::= **z** | **Z** | **?**

unbased_unsized_literal ::= **'0** | **'1** | 'z_or_x [10]

A.8.8 Strings

string_literal ::= **"** { Any_ASCII_Characters } **"**

A.9 General

A.9.1 Attributes

attribute_instance ::= **(*** attr_spec { **,** attr_spec } ***)**

attr_spec ::= attr_name [**=** constant_expression]

attr_name ::= identifier

A.9.2 Comments

comment ::=
      one_line_comment
    | block_comment
one_line_comment ::= **//** comment_text \n
block_comment ::= **/*** comment_text ***/**
comment_text ::= { Any_ASCII_character }

A.9.3 Identifiers

array_identifier ::= identifier
block_identifier ::= identifier
bin_identifier ::= identifier
c_identifier[2] ::= [**a-zA-Z_**] { [**a-zA-Z0-9_**] }
cell_identifier ::= identifier
class_identifier ::= identifier
class_variable_identifier ::= variable_identifier
clocking_identifier ::= identifier
config_identifier ::= identifier
const_identifier ::= identifier
constraint_identifier ::= identifier
covergroup_identifier ::= identifier
covergroup_variable_identifier ::= variable_identifier
cover_point_identifier ::= identifier
dynamic_array_variable_identifier ::= variable_identifier
enum_identifier ::= identifier
escaped_identifier ::= \ {any_ASCII_character_except_white_space} white_space
formal_identifier ::= identifier
function_identifier ::= identifier
generate_block_identifier ::= identifier
genvar_identifier ::= identifier
hierarchical_block_identifier ::= hierarchical_identifier
hierarchical_dynamic_array_variable_identifier ::= hierarchical_variable_identifier
hierarchical_event_identifier ::= hierarchical_identifier
hierarchical_identifier ::= [**$root .**] { identifier constant_bit_select **.** } identifier
hierarchical_net_identifier ::= hierarchical_identifier
hierarchical_parameter_identifier ::= hierarchical_identifier
hierarchical_task_identifier ::= hierarchical_identifier
hierarchical_tf_identifier ::= hierarchical_identifier
hierarchical_variable_identifier ::= hierarchical_identifier
identifier ::=

    simple_identifier
    | escaped_identifier
index_variable_identifier ::= identifier
interface_identifier ::= identifier
interface_instance_identifier ::= identifier
inout_port_identifier ::= identifier
input_port_identifier ::= identifier
instance_identifier ::= identifier
library_identifier ::= identifier
member_identifier ::= identifier
method_identifier ::= identifier
modport_identifier ::= identifier
module_identifier ::= identifier
net_identifier ::= identifier
output_port_identifier ::= identifier
package_identifier ::= identifier
package_scope ::=
        package_identifier **::**
    | **$unit ::**
parameter_identifier ::= identifier
port_identifier ::= identifier
production_identifier ::= identifier
program_identifier ::= identifier
property_identifier ::= identifier
ps_class_identifier ::= [package_scope] class_identifier
ps_covergroup_identifier ::= [package_scope] covergroup_identifier
ps_identifier ::= [package_scope] identifier
ps_or_hierarchical_net_identifier ::= [package_scope] net_identifier | hierarchical_net_identifier
ps_or_hierarchical_tf_identifier ::= [package_scope] tf_identifier | hierarchical_tf_identifier
ps_parameter_identifier ::=
        [package_scope] parameter_identifier
    | { generate_block_identifier [**[** constant_expression **]**] **.** } parameter_identifier
ps_property_identifier ::= [package_scope] property_identifier
ps_sequence_identifier ::= [package_scope] sequence_identifier
ps_type_identifier ::= [package_scope] type_identifier
sequence_identifier ::= identifier
signal_identifier ::= identifier
simple_identifier[2] ::= [**a-zA-Z_**] { [**a-zA-Z0-9_$**] }
specparam_identifier ::= identifier
system_tf_identifier[3] ::= **$**[**a-zA-Z0-9_$**]{ [**a-zA-Z0-9_$**] }
task_identifier ::= identifier
tf_identifier ::= identifier
terminal_identifier ::= identifier
topmodule_identifier ::= identifier

type_identifier ::= identifier

udp_identifier ::= identifier

variable_identifier ::= identifier

A.9.4 White space

white_space ::= space | tab | newline | eof[4]

A.10 Footnotes (normative)

1) Embedded spaces are illegal.

2) A simple_identifier, c_identifier, and arrayed_reference shall start with an alpha or underscore (_) character, shall have at least one character, and shall not have any spaces.

3) The $ character in a system_tf_identifier shall not be followed by white_space. A system_tf_identifier shall not be escaped.

4) End of file.

5) The unsigned number or fixed-point number in time_literal shall not be followed by a white_space.

6) implicit_class_handle shall only appear within the scope of a class_declaration or out-of-block method declaration.

7) In any one declaration, only one of **protected** or **local** is allowed, only one of **rand** or **randc** is allowed, and **static** and/or **virtual** can appear only once.

8) dpi_function_proto return types are restricted to small values, per 26.4.5.

9) Formals of dpi_function_proto and dpi_task_proto cannot use pass-by-reference mode, and class types cannot be passed at all; for the complete set of restrictions, see 26.4.6.

10) The apostrophe (′) in unbased_unsized_literal shall not be followed by white_space.

11) In packed_dimension, unsized_dimension is permitted only in declarations of import DPI functions; see dpi_function_proto.

12) When a packed dimension is used with the **struct** or **union** keyword, the **packed** keyword shall also be used.

13) A charge strength shall only be used with the **trireg** keyword. When the **vectored** or **scalared** keyword is used, there shall be at least one packed dimension.

14) In a data_declaration that is not within the procedural context, it shall be illegal to use the **automatic** keyword. In a data_declaration, it shall be illegal to omit the explicit data_type before a list_of_variable_decl_assignments unless the **var** keyword is used.

15) It shall be legal to omit the covergroup_variable_identifier from a covergroup instantiation only if this implicit instantiation is within a class that has no other instantiation of the covergroup.

16) The **.*** token shall appear at most once in a list of port connections.

17) A timeunits_declaration shall be legal as a non_port_module_item, non_port_interface_item, non_port_program_item, package_item, or class_item only if it repeats and matches a previous timeunits_declaration within the same time scope.

18) In a multiple_concatenation, it shall be illegal for the multiplier not to be a constant_expression unless the type of the concatenation is string.

19) In a shallow copy, the expression must evaluate to an object handle.

20) It shall be legal to use the **$** primary in an open_value_range of the form **[** expression **: $]** or **[$:** expression **]**.

21) **{ }** shall only be legal in the context of a queue.

22) The **$** primary shall be legal only in a select for a queue variable or in an open_value_range.

23) A type_identifier shall be legal as an enum_base_type if it denotes an integer_atom_type, with which an additional packed dimension is not permitted, or an integer_vector_type.

24) In a constant_function_call, all arguments shall be constant_expressions.

25) The list_of_port_declarations syntax is explained in 19.8, which also imposes various semantic restrictions, e.g., a **ref** port must be of a variable type and an **inout** port must not be. It shall be illegal to initialize a port that is not a variable **output** port.

26) It shall be legal to declare a **void** struct_union_member only within tagged unions.

27) An expression that is used as the argument in a type_reference shall not contain any hierarchical references or references to elements of dynamic objects.

28) When a type_reference is used in a net declaration, it shall be preceded by a net type keyword; and when it is used in a variable declaration, it shall be preceded by the **var** keyword.

29) It shall be legal to use a type_reference constant_primary as the casting_type in a static cast. It shall be illegal for a type_reference constant_primary to be used with any operators except the equality/ inequality and case equality/inequality operators.

30) A streaming_concatenation expression shall not be nested within another variable_lvalue. A streaming_concatenation shall not be the target of the increment or decrement operator nor the target of any assignment operator except the simple (**=**) or nonblocking assignment (**<=**) operator.

31) Within an interface_declaration, it shall only be legal for a module_or_interface_or_generate_item to be an interface_or_generate_item. Within a module_declaration, except when also within an interface_declaration, it shall only be legal for a module_or_interface_or_generate_item to be a module_or_generate_item.

32) A genvar_identifier shall be legal in a constant_primary only within a genvar_expression.

33) When a net_port_type contains a data_type, it shall only be legal to omit the explicit net_type when declaring an **inout** port.

34) In a tf_port_item, it shall be illegal to omit the explicit port_identifier except within a function_prototype or task_prototype.

35) In a constant_assignment_pattern_expression, all member expressions shall be constant expressions.

36) It shall be illegal to omit the parentheses in a tf_call unless the subroutine is a task, void function, or class method. If the subroutine is a nonvoid class function method, it shall be illegal to omit the parentheses if the call is directly recursive.

37) It shall be illegal for a program_generate_item to include any item that would be illegal in a program_declaration outside of a program_generate_item.

Annex B

(normative)

Keywords

SystemVerilog reserves the keywords listed in Table B-1.

Legend:

— * indicates SystemVerilog reserved words that are not reserved in IEEE Std 1364.

Table B-1—Reserved keywords

alias*	endmodule	matches*	small
always	endpackage*	medium	solve*
always_comb*	endprimitive	modport*	specify
always_ff*	endprogram*	module	specparam
always_latch*	endproperty*	nand	static*
and	endspecify	negedge	string*
assert*	endsequence*	new*	strong0
assign	endtable	nmos	strong1
assume*	endtask	nor	struct*
automatic	enum*	noshowcancelled	super*
before*	event	not	supply0
begin	expect*	notif0	supply1
bind*	export*	notif1	table
bins*	extends*	null*	tagged*
binsof*	extern*	or	task
bit*	final*	output	this*
break*	first_match*	package*	throughout*
buf	for	packed*	time
bufif0	force	parameter	timeprecision*
bufif1	foreach*	pmos	timeunit*
byte*	forever	posedge	tran
case	fork	primitive	tranif0
casex	forkjoin*	priority*	tranif1
casez	function	program*	tri
cell	generate	property*	tri0
chandle*	genvar	protected*	tri1
class*	highz0	pull0	triand
clocking*	highz1	pull1	trior
cmos	if	pulldown	trireg
config	iff*	pullup	type*
const*	ifnone	pulsestyle_onevent	typedef*
constraint*	ignore_bins*	pulsestyle_ondetect	union*
context*	illegal_bins*	pure*	unique*
continue*	import*	rand*	unsigned
cover*	incdir	randc*	use
covergroup*	include	randcase*	uwire
coverpoint*	initial	randsequence*	var*
cross*	inout	rcmos	vectored
deassign	input	real	virtual*
default	inside*	realtime	void*
defparam	instance	ref*	wait
design	int*	reg	wait_order*
disable	integer	release	wand
dist*	interface*	repeat	weak0
do*	intersect*	return*	weak1
edge	join	rnmos	while
else	join_any*	rpmos	wildcard*
end	join_none*	rtran	wire
endcase	large	rtranif0	with*
endclass*	liblist	rtranif1	within*
endclocking*	library	scalared	wor
endconfig	local*	sequence*	xnor
endfunction	localparam	shortint*	xor
endgenerate	logic*	shortreal*	
endgroup*	longint*	showcancelled	
endinterface*	macromodule	signed	

Annex C

(normative)

Std package

C.1 General

The standard package contains system types (see 8.10.1). The following types are provided by the std built-in package. The descriptions of the semantics of these types are defined in the indicated subclauses.

C.2 Semaphore

The semaphore class is described in 14.2, and its prototype is as follows:

```
class semaphore;
   function new(int keyCount = 0);
   task put(int keyCount = 1);
   task get(int keyCount = 1);
   function int try_get(int keyCount = 1);
endclass
```

C.3 Mailbox

The mailbox class is described in 14.3, and its prototype is as follows:

The *dynamic_singular_type* below represents a special type that enables run-time type checking.

```
class mailbox #(type T = dynamic_singular_type) ;
   function new(int bound = 0);
   function int num();
   task put( T message);
   function int try_put( T message);
   task get( ref T message );
   function int try_get( ref T message );
   task peek( ref T message );
   function int try_peek( ref T message );
endclass
```

C.4 Randomize

The randomize function is described in 13.11, and its prototype is as follows:

```
function int randomize( ... );
```

The syntax for the randomize function is as follows:

```
randomize( variable_identifier {, variable_identifier } )
   [ with constraint_block ];
```

C.5 Process

The process class is described in [11.9](#), and its prototype is as follows:

```
class process;
    enum state { FINISHED, RUNNING, WAITING, SUSPENDED, KILLED };

    static function process self();
    function state status();
    task kill();
    task await();
    task suspend();
    task resume();
endclass
```

Annex D

(normative)

Linked lists

D.1 General

The List package implements a classic list data structure and is analogous to the standard template library (STL) List container that is popular with C++ programmers. The container is defined as a parameterized class; in other words, it can be customized to hold data of any type.

D.2 List definitions

list: A doubly linked list, where every element has a predecessor and successor. A list is a sequence that supports both forward and backward traversal, as well as amortized constant time insertion and removal of elements at the beginning, end, or middle.

container: A collection of data of the same type. Containers are objects that contain and manage other data. Containers provide an associated iterator that allows access to the contained data.

iterator: An object that represents a position of an element in a container. Objects play a role similar to that of an array subscript and allow users to access a particular element and to traverse through the container.

D.3 List declaration

The List package supports lists of any arbitrary predefined type, such as **integer**, **string**, or class object.

Any iterator that refers to the position of an element that is removed from a list becomes invalid and thus unable to iterate over the list.

To declare a specific list, users must first include the generic List class declaration from the standard include area and then declare the specialized list type:

```
`include <List.vh>
...
List#(T) dl;      // dl is a List of 'T' elements
```

D.3.1 Declaring list variables

List variables are declared by providing a specialization of the generic List class:

```
List#(integer) il;          // Object il is a list of integer
typedef List#(Packet) PList;   // Class Plist is a list of Packet objects
```

The List specialization declares a list of the indicated type. The type used in the list declaration determines the type of the data stored in the list elements.

D.3.2 Declaring list iterators

List iterators are declared by providing a specialization of the generic List_Iterator class:

```
List_Iterator#(string) s;        // Object s is a list-of-string iterator
List_Iterator#(Packet) p, q;     // p and q are iterators to a list-of-Packet
```

D.4 Linked list class prototypes

The class prototypes in D.4.1 and D.4.2 describe the generic List and List_Iterator classes. Only the public interface is included here.

D.4.1 List_Iterator class prototype

```
class List_Iterator#(parameter type T);
    extern function void next();
    extern function void prev();
    extern function int neq( List_Iterator#(T) iter );
    extern function int eq( List_Iterator#(T) iter );
    extern function T data();
endclass
```

D.4.2 List class prototype

```
class List#(parameter type T);
    extern function new();
    extern function int size();
    extern function int empty();
    extern function void push_front( T value );
    extern function void push_back( T value );
    extern function T front();
    extern function T back();
    extern function void pop_front();
    extern function void pop_back();
    extern function List_Iterator#(T) start();
    extern function List_Iterator#(T) finish();
    extern function void insert( List_Iterator#(T) position, T value );
    extern function void insert_range( List_Iterator#(T) position,
                                        first, last );
    extern function void erase( List_Iterator#(T) position );
    extern function void erase_range( List_Iterator#(T) first, last );
    extern function void set( List_Iterator#(T) first, last );
    extern function void swap( List#(T) lst );
    extern function void clear();
    extern function void purge();
endclass
```

D.5 List_Iterator methods

The List_Iterator class provides methods to iterate over the elements of lists. These methods are described in D.5.1 through D.5.5.

D.5.1 Next()

```
function void next();
```

The `next()` method changes the iterator so that it refers to the next element in the list.

D.5.2 Prev()

```
function void prev();
```

The `prev()` method changes the iterator so that it refers to the previous element in the list.

D.5.3 Eq()

```
function int eq( List_Iterator#(T) iter );
```

The `eq()` method compares two iterators and returns `1` if both iterators refer to the same list element. Otherwise, it returns `0`.

```
if( i1.eq(i2) ) $display("both iterators refer to the same element");
```

D.5.4 Neq()

```
function int neq( List_Iterator#(T) iter );
```

The `neq()` method is the negation of `eq()`; it compares two iterators and returns `0` if both iterators refer to the same list element. Otherwise, it returns `1`.

D.5.5 Data()

```
function T data();
```

The `data()` method returns the data stored in the element at the given iterator location.

D.6 List methods

The List class provides methods to query the size of the list; obtain iterators to the head or tail of the list; retrieve the data stored in the list; and methods to add, remove, and reorder the elements of the list.

D.6.1 Size()

```
function int size();
```

The `size()` method returns the number of elements stored in the list.

```
while ( list1.size > 0 ) begin   // loop while still elements in the list
    ...
end
```

D.6.2 Empty()

```
function int empty();
```

The `empty()` method returns 1 if the number elements stored in the list is zero and 0 otherwise.

```
if ( list1.empty )
    $display( "list is empty" );
```

D.6.3 Push_front()

```
function void push_front( T value );
```

The `push_front()` method inserts the specified value at the front of the list.

```
List#(int) numbers;
numbers.push_front(10);
numbers.push_front(5);     // numbers contains { 5 , 10 }
```

D.6.4 Push_back()

```
function void push_back( T value );
```

The `push_back()` method inserts the specified value at the end of the list.

```
List#(string) names;
names.push_back("Donald");
names.push_back("Mickey");    // names contains { "Donald", "Mickey" }
```

D.6.5 Front()

```
function T front();
```

The `front()` method returns the data stored in the first element of the list (valid only if the list is not empty).

D.6.6 Back()

```
function T back();
```

The `back()` method returns the data stored in the last element of the list (valid only if the list is not empty).

```
List#(int) numbers;
numbers.push_front(3);
numbers.push_front(2);
numbers.push_front(1);
$display( numbers.front, numbers.back );  // displays 1 3
```

D.6.7 Pop_front()

```
function void pop_front();
```

The `pop_front()` method removes the first element of the list. If the list is empty, this method is illegal and can generate an error.

D.6.8 Pop_back()

```
function void pop_back();
```

The `pop_back()` method removes the last element of the list. If the list is empty, this method is illegal and can generate an error.

```
while ( lp.size > 1 ) begin   // remove all but the center element from
                              // an odd-sized list lp
   lp.pop_front();
   lp.pop_back();
end
```

D.6.9 Start()

```
function List_Iterator#(T) start();
```

The `start()` method returns an iterator to the position of the first element in the list.

D.6.10 Finish()

```
function List_Iterator#(T) finish();
```

The `finish()` method returns an iterator to a position just past the last element in the list. The last element in the list can be accessed using `finish.prev`.

```
List#(int) lst;      // display contents of list lst in position order
for ( List_Iterator#(int) p = lst.start; p.neq(lst.finish); p.next )
   $display( p.data );
```

D.6.11 Insert()

```
function void insert( List_Iterator#(T) position, T value );
```

The `insert()` method inserts the given `data` (value) into the list at the position specified by the iterator (before the element, if any, that was previously at the iterator's position). If the iterator is not a valid position within the list, then this operation is illegal and can generate an error.

```
function void add_sort( List#(byte) L, byte value );
   for ( List_Iterator#(byte) p = L.start; p.neq(L.finish) ; p.next )
      unique case (1)
         p.data < value : continue ;
         p.data == value : return ;
         p.data > value : break ;
      endcase
   lst.insert( p, value ); // Add to sorted list (ascending order)
endfunction: add_sort
```

D.6.12 Insert_range()

```
function void insert_range( List_Iterator#(T) position, first, last );
```

The `insert_range()` method inserts the elements contained in the list range specified by the iterators first and last at the specified list position (before the element, if any, that was previously at the position iterator). All the elements from first up to, but not including, last are inserted into the list. If the last iterator refers to an element before the first iterator, the range wraps around the end of the list. The range iterators can specify a range either in another list or in the same list as being inserted.

If the position iterator is not a valid position within the list or if the range iterators are invalid (i.e., they refer to different lists or to invalid positions), then this operation is illegal and can generate an error.

D.6.13 Erase()

```
function void erase( List_Iterator#(T) position );
```

The `erase()` method removes from the list the element at the specified position. After `erase()` returns, the position iterator becomes invalid.

```
list1.erase( list1.start );   // same as pop_front
```

If the position iterator is not a valid position within the list, this operation is illegal and can generate an error.

D.6.14 Erase_range()

```
function void erase_range( List_Iterator#(T) first, last );
```

The `erase_range()` method removes from a list the range of elements specified by the first and last iterators. This operation removes elements from the first iterator's position up to, but not including, the last iterator's position. If the last iterator refers to an element before the first iterator, the range wraps around the end of the list.

```
list1.erase_range( list1.start, list1.finish ); // Remove all elements from
                                                 // list1
```

If the range iterators are invalid (i.e., they refer to different lists or to invalid positions), then this operation is illegal and can generate an error.

D.6.15 Set()

```
function void set( List_Iterator#(T) first, last );
```

The `set()` method assigns to the list object the elements that lie in the range specified by the first and last iterators. After this method returns, the modified list shall have a size equal to the range specified by first and last. This method copies the data from the first iterator's position up to, but not including, the last iterator's position. If the last iterator refers to an element before the first iterator, the range wraps around the end of the list.

```
list2.set( list1.start, list2.finish );     // list2 is a copy of list1
```

If the range iterators are invalid (i.e., they refer to different lists or to invalid positions), then this operation is illegal and can generate an error.

D.6.16 Swap()

```
function void swap( List#(T) lst );
```

The `swap()` method exchanges the contents of two equal-size lists.

```
list1.swap( list2 ); // swap the contents of list1 to list2 and vice versa
```

Swapping a list with itself has no effect. If the lists are of different sizes, this method can issue a warning.

D.6.17 Clear()

```
function void clear();
```

The `clear()` method removes all the elements from a list, but not the list itself (i.e., the list header itself).

```
list1.clear();    // list1 becomes empty
```

D.6.18 Purge()

```
function void purge();
```

The `purge()` method removes all the list elements (as in clear) and the list itself. This accomplishes the same effect as assigning null to the list. A purged list must be recreated using **new** before it can be used again.

```
list1.purge();    // same as list1 = null
```

Annex E

(normative)

Formal semantics of concurrent assertions

E.1 Introduction

This annex presents a formal semantics for SystemVerilog concurrent assertions. Immediate assertions and coverage statements are not discussed here. Throughout this annex, "assertion" is used to mean "concurrent assertion". The semantics is defined by a relation that determines when a finite or infinite word (i.e., trace) satisfies an assertion. Intuitively, such a word represents a sequence of valuations of SystemVerilog variables sampled at the finest relevant granularity of time (e.g., at the granularity of simulator cycles). The process by which such words are produced is closely related to the SystemVerilog scheduling semantics and is not defined here. In this annex, words are assumed to be sequences of elements, each element being either a set of atomic propositions or one of two special symbols used as placeholders when extending finite words. The atomic propositions are not further defined. The meaning of satisfaction of a SystemVerilog boolean expression by a set of atomic propositions is assumed to be understood.

The semantics is based on an abstract syntax for SystemVerilog assertions. There are several advantages to using the abstract syntax rather than the full SystemVerilog assertions BNF.

a)   The abstract syntax facilitates separation of derived operators from basic operators. The satisfaction relation is defined explicitly only for assertions built from basic operators.

b)   The abstract syntax avoids reliance on operator precedence, associativity, and auxiliary rules for resolving syntactic and semantic ambiguities.

c)   The abstract syntax simplifies the assertion language by eliminating some features that tend to encumber the definition of the formal semantics.

1)   The abstract syntax eliminates local variable declarations. The semantics of local variables is written with implicit types.

2)   The abstract syntax eliminates instantiation of sequences and properties. The semantics of an assertion with an instance of a sequence or nonrecursive property is the same as the semantics of a related assertion obtained by replacing the sequence or nonrecursive property instance with an explicitly written sequence or property. The explicit sequence or property is obtained from the body of the associated declaration by substituting actual arguments for formal arguments. A separate subclause defines the semantics of instances of recursive properties in terms of the semantics of instances of nonrecursive properties.

3)   The abstract syntax does not allow implicit clocks. Clocking event controls must be applied explicitly in the abstract syntax.

4)   The abstract syntax does not allow explicit procedural enabling conditions for assertions. Procedural enabling conditions are utilized in the semantics definition (see 4.3.1), but the method for extracting such conditions is not defined in this annex.

d)   The abstract syntax eliminates the distinction between *property_expr* and *property_spec* from the full BNF. Without the distinction, **disable iff** is a general, nestable property-building operator, while in the full BNF **disable iff** can be attached only at the top level of a property. Semantically, there is no need for this restriction on the placement of **disable iff**. The abstract syntax thus eliminates an unnecessary semantic layer while maintaining the simple inductive form for the definition of the semantics of properties. As a result, semantics is given for some properties that do not correspond to forms from the full BNF, but this does not degrade the definitions for the properties that do correspond to forms from the full BNF.

In order to use this annex to determine the semantics of a SystemVerilog assertion, the assertion must first be transformed into an enabling condition together with an assertion in the abstract syntax. For assertions that do not involve recursive properties, this transformation involves eliminating sequence and nonrecursive property instances by substitution, eliminating local variable declarations, introducing parentheses, determining the enabling condition, determining implicit or inferred clocking event controls, and eliminating redundant clocking event controls. For example, the following SystemVerilog assertion

```
sequence s(x,y); x ##1 y; endsequence
sequence t(z); @(c) z[*1:2] ##1 B; endsequence
always @(c) if (b) assert property (s(A,B) |=> t(A));
```

is transformed into the enabling condition "*b*" together with the assertion

```
always @(c) assert property ((A ##1 B) |=> (A[*1:2] ##1 B))
```

in the abstract syntax.

If the SystemVerilog assertion involves instances of recursive properties, then the transformation replaces these instances with placeholder functions of the actual arguments. The semantics of an instance of a recursive property is defined in terms of associated nonrecursive properties in E.5. Once the semantics of the recursive property instances is understood, the placeholder functions are treated as properties with this semantics. Then the ordinary definitions can be applied to the transformed assertion in the abstract syntax together with placeholder functions.

E.2 Abstract syntax

E.2.1 Abstract grammars

In the following abstract grammars, *b* denotes a boolean expression, *v* denotes a local variable name, and *e* denotes an expression.

The abstract grammar for unclocked sequences is

```
R ::= b                      // "boolean expression" form
    | ( 1, v = e )           // "local variable sampling" form
    | ( R )                  // "parenthesis" form
    | ( R ##1 R )            // "concatenation" form
    | ( R ##0 R )            // "fusion" form
    | ( R or R )             // "or" form
    | ( R intersect R )      // "intersect" form
    | first_match ( R )      // "first match" form
    | R [ *0 ]               // "null repetition" form
    | R [ *1:$ ]             // "unbounded repetition" form
```

The abstract grammar for clocked sequences is

```
S ::= @(b) R                              // "clock" form
    | ( S )                               // "parenthesized" form
    | ( S ##1 S )                         // "concatenation" form
```

The abstract grammar for unclocked properties is

```
P ::= R                    // "sequence" form
    | ( P )                // "parenthesis" form
```

```
    | not P                      // "negation" form
    | ( P or P )                 // "or" form
    | ( P and P )                // "and" form
    | ( R |-> P )                // "implication" form
    | disable iff ( b ) P        // "reset" form
```

Each instance of R in this production must be a nondegenerate unclocked sequence. In the "sequence" form, R must not be tightly satisfied by the empty word. See E.3.2 and E.3.5 for the definitions of nondegeneracy and tight satisfaction.

The abstract grammar for clocked properties is

```
Q ::= @( b ) P              // "clock" form
    | S                     // "sequence" form
    | ( Q )                 // "parenthesis" form
    | not Q                 // "negation" form
    | ( Q or Q )            // "or" form
    | ( Q and Q )           // "and" form
    | ( S |-> Q )           // "implication" form
    | disable iff ( b ) Q   // "reset" form
```

Each instance of S in this production must be a nondegenerate clocked sequence. In the "sequence" form, S must not be tightly satisfied by the empty word. See E.3.2 and E.3.5 for the definitions of nondegeneracy and tight satisfaction.

The abstract grammar for assertions is

```
A ::= always assert property Q              // "always" form
    | always @( b) assert property P        // "always with clock" form
    | initial assert property Q             // "initial" form
    | initial @( b) assert property P       // "initial with clock" form
```

E.2.2 Notations

In E.2.3, the following notational conventions will be used: b and c denote boolean expressions; v denotes a local variable name; e denotes an expression; R, R_1, and R_2 denote unclocked sequences; S, S_1, and S_2 denote clocked sequences; P, P_1, and P_2 denote unclocked properties; Q denotes a clocked property; A denotes an assertion; i, j, k, m, and n denote non-negative integer constants.

E.2.3 Derived forms

Internal parentheses are omitted in compositions of the (associative) operators **##1** and **or**.

E.2.3.1 Derived nonoverlapping implication operator

— $(R_1 |=> P) \equiv ((R_1 \#\#1 \ 1) |-> P)$.

— $(S_1 |=> Q) \equiv ((S_1 \#\#1 \ @(1) \ 1) |-> Q)$.

E.2.3.2 Derived consecutive repetition operators

— Let $m > 0$. $R [*m] \equiv (R \#\#1 \ R \#\#1 \ \cdots \#\#1 \ R)$ // m copies of R.

— $R [*0:\$] \equiv (R [*0] \ \text{or} \ R [*1:\$])$.

— Let $m \leq n$. $R [*m:n] \equiv (R [*m] \ \text{or} \ R [*m+1] \ \text{or} \ \cdots \ \text{or} \ R [*n])$.

— Let $m > 1$. $R [*m:\$] \equiv (R [*m-1] \ \#\#1 \ R [*1:\$])$.

E.2.3.3 Derived delay and concatenation operators

Let $m \le n$.

— $(\,\#\#[m{:}n]\ R\,) \equiv (1[*m{:}n]\ \#\#1\ R\,)$.

— $(\,\#\#[m{:}\$]\ R\,) \equiv (1[*m{:}\$]\ \#\#1\ R\,)$.

— $(\,\#\#m\ R\,) \equiv (1[*m]\ \#\#1\ R\,)$.

— Let $m > 0$. $(\,R_1\ \#\#[m{:}n]\ R_2\,) \equiv (\,R_1\ \#\#1\ 1[*m-1{:}n-1]\ \#\#1\ R_2\,)$.

— Let $m > 0$. $(\,R_1\ \#\#[m{:}\$]\ R_2\,) \equiv (\,R_1\ \#\#1\ 1[*m-1{:}\$]\ \#\#1\ R_2\,)$.

— Let $m > 1$. $(\,R_1\ \#\#m\ R_2\,) \equiv (\,R_1\ \#\#1\ 1[*m-1]\ \#\#1\ R_2\,)$.

— $(\,R_1\ \#\#[0{:}0]\ R_2\,) \equiv (\,R_1\ \#\#0\ R_2\,)$.

— Let $n > 0$. $(\,R_1\ \#\#[0{:}n]\ R_2\,) \equiv ((\,R_1\ \#\#0\ R_2\,)\ \textbf{or}\ (\,R_1\ \#\#[1{:}n]\ R_2\,))$.

— $(\,R_1\ \#\#[0{:}\$]\ R_2\,) \equiv ((\,R_1\ \#\#0\ R_2\,)\ \textbf{or}\ (\,R_1\ \#\#[1{:}\$]\ R_2\,))$.

E.2.3.4 Derived nonconsecutive repetition operators

Let $m \le n$.

— $b\ [{-}{>}m{:}n] \equiv (\,!b\ [*0{:}\$]\ \#\#1\ b\,)[*m{:}n]$.

— $b\ [{-}{>}m{:}\$] \equiv (\,!b\ [*0{:}\$]\ \#\#1\ b\,)[*m{:}\$]$.

— $b\ [{-}{>}m] \equiv (\,!b\ [*0{:}\$]\ \#\#1\ b)[*m]$.

— $b\ [=m{:}n] \equiv (\,b\ [{-}{>}m{:}n]\ \#\#1\ !b\ [*0{:}\$]\,)$.

— $b\ [=m{:}\$] \equiv (\,b\ [{-}{>}m{:}\$]\ \#\#1\ !b\ [*0{:}\$]\,)$.

— $b\ [=m] \equiv (\,b\ [{-}{>}m]\ \#\#1\ !b\ [*0{:}\$]\,)$.

E.2.3.5 Other derived operators

— $(\,R_1\ \textbf{and}\ R_2\,)$
$\equiv (((\,R_1\ \#\#1\ 1[*0{:}\$])\ \textbf{intersect}\ R_2\,)\ \textbf{or}\ (\,R_1\ \textbf{intersect}\ (\,R_2\ \#\#1\ 1[*0{:}\$])))$.

— $(\,R_1\ \textbf{within}\ R_2\,) \equiv ((1[*0{:}\$]\ \#\#1\ R_1\ \#\#1\ 1[*0{:}\$])\ \textbf{intersect}\ R_2\,)$.

— $(\,b\ \textbf{throughout}\ R\,) \equiv ((\,b\ [*0{:}\$])\ \textbf{intersect}\ R\,)$.

— $(\,R,\ v = e\,) \equiv (\,R\ \#\#0\ (1,\ v = e\,))$.

— $(\,R,\ v_1 = e_1, \dots, v_k = e_k\,) \equiv ((\,R, v_1 = e_1\,)\ \#\#0\ (1,\ v_2 = e_2, \dots, v_k = e_k\,))\ \textbf{for}\ k > 1$

— $(\,\textbf{if}\,(\,b\,)\,P\,) \equiv (\,b\,|{-}{>}P\,)$

— $(\,\textbf{if}\,(\,b\,)\,P_1\ \textbf{else}\ P_2\,) \equiv ((\,b\ |{-}{>}P_1\,)\ \textbf{and}\,(\,!b\ |{-}{>}P_2\,))$

E.3 Semantics

Let **P** be the set of atomic propositions.

The semantics of assertions and properties is defined via a relation of satisfaction by empty, finite, and infinite words over the alphabet $\Sigma = 2^{\textbf{P}} \cup \{\top, \bot\}$. Such a word is an empty, finite, or infinite sequence of elements of Σ. The number of elements in the sequence is called the *length* of the word, and the length of word w is denoted $|w|$, where $|w|$ is either a non-negative integer or infinity.

The sequence elements of a word are called its *letters* and are assumed to be indexed consecutively beginning at zero. If $|w| > 0$, then the first letter of w is denoted w^0; if $|w| > 1$, then the second letter of w is denoted w^1; and so forth. $w^{i..}$ denotes the word obtained from w by deleting its first i letters. If $i < |w|$, then $w^{i..} = w^i w^{i+1} \dots$. If $i \ge |w|$, then $w^{i..}$ is empty.

If $i \leq j$, then $w^{i,j}$ denotes the finite word obtained from w by deleting its first i letters and also deleting all letters after its $(j + 1)$st. If $i \leq j < |w|$, then $w^{i,j} = w^i w^{i+1}...w^j$.

If w is a word over Σ, define $\overline{w}$ to be the word obtained from w by interchanging T with $\perp$. More precisely, $\overline{w}^i = \mathsf{T}$ if $w^i = \perp$; $\overline{w}^i = \perp$ if $w^i = \mathsf{T}$; and $\overline{w}^i = w^i$ if w^i is an element in $2^{\mathbf{P}}$.

The semantics of clocked sequences and properties is defined in terms of the semantics of unclocked sequences and properties. See E.3.1.

It is assumed that the satisfaction relation $\zeta \models b$ is defined for elements ζ in $2^{\mathbf{P}}$ and boolean expressions b. For any boolean expression b, define

$$\mathsf{T} \models b \quad \text{and} \quad \perp \not\models b .$$

E.3.1 Rewrite rules for clocks

The semantics of clocked sequences and properties is defined in terms of the semantics of unclocked sequences and properties. The following rewrite rules define the transformation of a clocked sequence or property into an unclocked version that is equivalent for the purposes of defining the satisfaction relation. In this transformation, it is required that the conditions in event controls not be dependent upon any local variables.

— `@(c) b` $\longmapsto$ `( !c [*0:$] ##1 c & b )` .

— `@(c)( 1, v = e )` $\longmapsto$ `( @(c) 1 ##0 ( 1, v = e ))` .

— `@(c)( P )` $\longmapsto$ `( @(c) P )`.

— `@(c)( ` R_1 `##1 ` R_2 ` )` $\longmapsto$ `( @(c) ` R_1 ` ##1 @(c) ` R_2 ` )` .

— `@(c)( ` R_1 `##0 ` R_2 ` )` $\longmapsto$ `( @(c) ` R_1 ` ##0 @(c) ` R_2 ` )` .

— `@(c)( ` R_1 ` or ` R_2 ` )` $\longmapsto$ `( @(c) ` R_1 ` or @(c) ` R_2 ` )` .

— `@(c)( ` R_1 ` intersect ` R_2 ` )` $\longmapsto$ `( @(c) ` R_1 ` intersect @(c) ` R_2 ` )` .

— `@(c) first_match ( R )` $\longmapsto$ `first_match ( @(c) R )` .

— `@(c) R [*0]` $\longmapsto$ `( @(c) R ) [*0]` .

— `@(c) R [*1:$]` $\longmapsto$ `( @(c) R ) [*1:$]` .

— `@(c) disable iff ( b ) P` $\longmapsto$ `disable iff ( b ) @(c) P` .

— `@(c) not P` $\longmapsto$ `not @(c) P`.

— `@(c)( R |-> P )` $\longmapsto$ `( @(c) R |-> @(c) P )` .

— `@(c)( ` P_1 ` or ` P_2 ` )` $\longmapsto$ `( @(c) ` P_1 ` or @(c) ` P_2 ` )`.

— `@(c)( ` P_1 ` and ` P_2 ` )` $\longmapsto$ `( @(c) ` P_1 ` and @(c) ` P_2 ` )`.

E.3.2 Tight satisfaction without local variables

Tight satisfaction is denoted by $\models$. For unclocked sequences without local variables, tight satisfaction is defined as follows: w, x, y, and z denote finite words over Σ.

— $w \models b$ iff $|w| = 1$ and $w^0 \models b$.

— $w \models (R)$ iff $w \models R$.

— $w \models (R_1$ `##1` $R_2)$ iff there exist x, y so that $w = xy$ and $x \models R_1$ and $y \models R_2$.

— $w \models (R_1$ `##0` $R_2)$ iff there exist x, y, z so that $w = xyz$ and $|y| = 1$, and $xy \models R_1$ and $yz \models R_2$.

— $w \models (R_1$ `or` $R_2)$ iff either $w \models R_1$ or $w \models R_2$.

— $w \models (R_1$ `intersect` $R_2)$ iff both $w \models R_1$ and $w \models R_2$.

— $w \models$ `first_match` (R) iff both

     — $w \models R$ and

     — if there exist x, y so that $w = xy$ and $\overline{x} \models R$, then y is empty.

— $w \models R$ `[*0]` iff $|w| = 0$.

— $w \models R$ `[*1:$]` iff there exist words $w_1, w_2, ..., w_j$ ($j \geq 1$) so that $w = w_1 w_2 ... w_j$ and for every i so that $1 \leq \iota \leq j$, $w_i \models R$.

If S is a clocked sequence, then $w \models S$ iff $w \models S'$, where S' is the unclocked sequence that results from S by applying the rewrite rules.

An unclocked sequence R is nondegenerate iff there exists a nonempty finite word w over Σ so that $w \models R$. A clocked sequence S is nondegenerate iff the unclocked sequence S' that results from S by applying the rewrite rules is nondegenerate.

E.3.3 Satisfaction without local variables

E.3.3.1 Neutral satisfaction

w denotes a nonempty finite or infinite word over Σ. Assume that all properties, sequences, and unclocked property fragments do not involve local variables.

Neutral satisfaction of assertions is as follows:

For the definition of neutral satisfaction of assertions, b denotes the boolean expression representing the enabling condition for the assertion. Intuitively, b is derived from the conditions in the context of a procedural assertion, while b is "1" for a declarative assertion.

— $w, b \models$ `always` $@(c)$ `assert property` P iff for every $0 \leq i < |w|$ so that $\overline{w}^i \models c$ and $\overline{w}^i \models b$, $w^{i..} \models @(c)\ P$.

— $w, b \models$ `always assert property` Q iff for every $0 \leq i < |w|$, if $\overline{w}^i \models b$ then $w^{i..} \models Q$.

— $w, b \models$ `initial` $@(c)$ `assert property` P iff for every $0 \leq i < |w|$ so that $\overline{w}^{0,i} \models !c$ `[*0:$]` `##1` c and $\overline{w}^i \models b$, $w^{i..} \models @(c)\ P$.

— $w, b \models$ `initial assert property` Q iff (if $\overline{w}^0 \models b$ then $w \models Q$) .

Neutral satisfaction of properties is as follows:

— $w \models (P)$ `iff` $w \models P$.

— $w \models Q$ iff $w \models Q'$, where Q' is the unclocked property that results from Q by applying the rewrite rules.

— $w \models$ `disable iff` (b) P iff either $w \models P$ or there exists $0 \leq k < |w|$ so that $w^k \models b$ and $w^{0,k-1} \top^\omega \models P$. Here, $w^{0,-1}$ denotes the empty word.

— $w \models$ `not` P iff $\overline{w} \not\models P$.

— $w \models R$ iff there exists $0 \leq j < |w|$ so that $w^{0,j} \models R$.

— $w \models (R$ `|->` $P)$ iff for every $0 \leq j < |w|$ so that $\overline{w}^{0,j} \models R$, $w^{j..} \models P$.

— $w \models (P_1$ `or` $P_2)$ iff $w \models P_1$ or $w \models P_2$.

— $w \models (P_1$ `and` $P_2)$ iff $w \models P_1$ and $w \models P_2$.

Remark: Because w is nonempty, it can be proved that $w \models$ `not` b iff $w \models !b$.

E.3.3.2 Weak and strong satisfaction by finite words

This subclause defines weak and strong satisfaction, denoted $\models^-$ and $\models^+$ (respectively) of an assertion A by a finite (possibly empty) word w over Σ. These relations are defined in terms of the relation of neutral satisfaction by infinite words as follows:

— $w \models^- A$ iff $w\, \mathsf{T}^\omega \models A$.
— $w \models^+ A$ iff $w\bot^\omega \models A$.

A tool checking for satisfaction of A by the finite word w should return the following:

— "Holds strongly" if $w \models^+ A$.
— "Fails" if $w \not\models^- A$.
— "Holds (but does not hold strongly)" if $w \models A$ and $w \not\models^+ A$.
— "Pending" if $w \models^- A$ and $w \not\models A$.

E.3.4 Local variable flow

This subclause defines inductively how local variable names flow through unclocked sequences. Below, "$\cup$" denotes set union, "$\cap$" denotes set intersection, "$-$" denotes set difference, and "$\{\}$" denotes the empty set.

The function "*sample*" takes a sequence as input and returns a set of local variable names as output. Intuitively, this function returns the set of local variable names that are sampled (i.e., assigned) in the sequence.

The function "*block*" takes a sequence as input and returns a set of local variable names as output. Intuitively, this function returns the set of local variable names that are blocked from flowing out of the sequence.

The function "*flow*" takes a set X of local variable names and a sequence as input and returns a set of local variable names as output. Intuitively, this function returns the set of local variable names that flow out of the sequence given the set X of local variable names that flow into the sequence.

The function "*sample*" is defined by

— *sample* $(b) = \{\}$.
— *sample* $((\,\mathtt{1},\ v = e\,)) = \{v\}$.
— *sample* $((\,R\,)) = sample\,(R)$.
— *sample* $((\,R_1\ \mathtt{\#\#1}\ R_2\,)) = sample\,(R_1) \cup sample\,(R_2)$.
— *sample* $((\,R_1\ \mathtt{\#\#0}\ R_2\,)) = sample\,(R_1) \cup sample\,(R_2)$.
— *sample* $((\,R_1\ \mathbf{or}\ R_2\,)) = sample\,(R_1) \cup sample\,(R_2)$.
— *sample* $((\,R_1\ \mathbf{intersect}\ R_2\,)) = sample\,(R_1) \cup sample\,(R_2)$.
— *sample* $(\mathbf{first\_match}\,(\,R\,)) = sample\,(R)$.
— *sample* $(R\ [\mathtt{*0}]) = \{\}$.
— *sample* $(R\ [\mathtt{*1:\$}]) = sample\,(R)$.

The function "*block*" is defined by

— *block* $(b) = \{\}$.
— *block* $((\,\mathtt{1},\ v = e\,)) = \{\}$.
— *block* $((\,R\,)) = block\,(R)$.

- $block((R_1$ `##1` $R_2)) = (block(R_1) - flow(\{\}, R_2)) \cup block(R_2)$.
- $block((R_1$ `##0` $R_2)) = (block(R_1) - flow(\{\}, R_2)) \cup block(R_2)$.
- $block((R_1$ `or` $R_2)) = block(R_1) \cup block(R_2)$.
- $block((R_1$ `intersect` $R_2)) = block(R_1) \cup block(R_2) \cup (sample(R_1) \cap sample(R_2))$.
- $block(\textbf{first\_match}(R)) = block(R)$.
- $block(R\,[*0]) = \{\}$.
- $block(R\,[*1:\$]) = block(R)$.

The function "*flow*" is defined by

- $flow(X, b) = X$.
- $flow(X, (1,\ v = e)) = X \cup \{v\}$.
- $flow(X, (R)) = flow(X, R)$.
- $flow(X, (R_1$ `##1` $R_2)) = flow(flow(X, R_1), R_2)$.
- $flow(X, (R_1$ `##0` $R_2)) = flow(flow(X, R_1), R_2)$.
- $flow(X, (R_1$ `or` $R_2)) = flow(X, R_1) \cap flow(X, R_2)$.
- $flow(X, (R_1$ `intersect` $R_2)) = (flow(X, R_1) \cup flow(X, R_2)) - block((R_1$ `intersect` $R_2))$.
- $flow(X, \textbf{first\_match}(R)) = flow(X, R)$.
- $flow(X, R\,[*0]) = X$.
- $flow(X, R\,[*1:\$]) = flow(X, R)$.

Remark: It can be proved that $flow(X, R) = (X \cup flow(\{\}, R)) - block(R)$. It follows that $flow(\{\}, R)$ and $block(R)$ are disjoint. It can also be proved that $flow(\{\}, R)$ is a subset of $sample(R)$.

E.3.5 Tight satisfaction with local variables

A local variable context is a function that assigns values to local variable names. If L is a local variable context, then dom(L) denotes the set of local variable names that are in the domain of L. If $D \subseteq$ dom(L), then $L|_D$ means the local variable context obtained from L by restricting its domain to D.

In the presence of local variables, tight satisfaction is a four-way relation defining when a finite word w over the alphabet Σ together with an input local variable context L_0 satisfies an unclocked sequence R and yields an output local variable context L_1. This relation is denoted

$$w, L_0, L_1 \models R .$$

and is defined below. It can be proved that the definition guarantees that $w, L_0, L_1 \models R$ implies dom(L_1) $= flow(\mathrm{dom}(L_0), R)$.

- $w, L_0, L_1 \models (1,\ v = e)$ iff $|w| = 1$ and $w^0 \models 1$ and $L_1 = \{(v, e[L_0, w^0])\} \cup L_0|_D$, where $e[L_0, w^0]$ denotes the value obtained from e by evaluating first according to L_0 and second according to w^0 and $D = \mathrm{dom}(L_0) - \{v\}$. In case $w^0 \in \{\top, \bot\}$, $e[L_0, \top]$ and $e[L_0, \bot]$ can be any constant values of the type of e.
- $w, L_0, L_1 \models b$ iff $|w| = 1$ and $w^0 \models b[L_0]$ and $L_1 = L_0$. Here $b[L_0]$ denotes the expression obtained from b by substituting values from L_0 .
- $w, L_0, L_1 \models (R)$ iff $w, L_0, L_1 \models R$.
- $w, L_0, L_1 \models (R_1$ `##1` $R_2)$ iff there exist x, y, L' so that $w = xy$ and $x, L_0, L' \models R_1$ and $y, L', L_1 \models R_2$.
- $w, L_0, L_1 \models (R_1$ `##0` $R_2)$ iff there exist x, y, z, L' so that $w = xyz$ and $|y| = 1$, and $xy, L_0, L' \models R_1$ and $yz, L', L_1 \models R_2$.

— $w, L_0, L_1 \models (\ R_1 \ \mathbf{or} \ R_2\ )$ iff there exists L' so that both of the following hold:

  — Either $w, L_0, L' \models R_1$ or $w, L_0, L' \models R_2$, and

  — $L_1 = L'|_D$, where $D = flow\,(\mathrm{dom}(L_0),\ (\ R_1 \ \mathbf{or} \ R_2\ ))$.

— $w, L_0, L_1 \models (\ R_1 \ \mathbf{intersect} \ R_2\ )$ iff there exist L', L'' so that $w, L_0, L' \models R_1$ and $w, L_0, L'' \models R_2$ and $L_1 = L'|_{D'} \cup L''|_{D''}$, where

$$D' = flow\,(\mathrm{dom}(L_0), R_1) - (block\,((\ R_1 \ \mathbf{intersect} \ R_2\ )) \cup sample\,(R_2))$$
$$D'' = flow\,(\mathrm{dom}(L_0), R_2) - (block\,((\ R_1 \ \mathbf{intersect} \ R_2\ )) \cup sample\,(R_1))$$

Remark: It can be proved that if $w, L_0, L' \models R_1$ and $w, L_0, L'' \models R_2$, then $L'|_{D'} \cup L''|_{D''}$ is a function.

— $w, L_0, L_1 \models \mathbf{first\_match}\,(\ R\ )$ iff both

  — $w, L_0, L_1 \models R$ and

  — If there exist x, y, L' so that $w = xy$ and $\bar{x}, L_0, L' \models R$, then y is empty.

— $w, L_0, L_1 \models R\ [\ast 0]$ iff $|w| = 0$ and $L_1 = L_0$.

— $w, L_0, L_1 \models R\ [\ast 1:\$]$ iff there exist $L_{(0)} = L_0, w_1, L_{(1)}, w_2, L_{(2)},..., w_j, L_{(j)} = L_1\ (j \geq 1)$ so that $w = w_1 w_2 ... w_j$ and for every i so that $1 \leq i \leq j$, $w_i, L_{(i-1)}, L_{(i)} \models R$.

If S is a clocked sequence, then $w, L_0, L_1 \models S$ iff $w, L_0, L_1 \models S'$, where S' is the unclocked sequence that results from S by applying the rewrite rules.

An unclocked sequence R is nondegenerate iff there exist a nonempty finite word w over Σ and local variable contexts L_0, L_1 so that $w, L_0, L_1 \models R$. A clocked sequence S is nondegenerate iff the unclocked sequence S' that results from S by applying the rewrite rules is nondegenerate.

E.3.6 Satisfaction with local variables

E.3.6.1 Neutral satisfaction

w denotes a nonempty finite or infinite word over Σ. L_0 and L_1 denote local variable contexts.

The rules defining neutral satisfaction of an assertion are identical to those without local variables, but with the understanding that the underlying properties can have local variables.

Neutral satisfaction of properties is as follows:

— $w \models Q$ iff $w, \{\} \models Q$.

— $w, L_0 \models Q$ iff $w, L_0 \models Q'$, where Q' is the unclocked property that results from Q by applying the rewrite rules.

— $w, L_0 \models \mathbf{disable\ iff}\ (b)\ P$ iff either $w, L_0 \models P$ or there exists $0 \leq k < |w|$ so that $w^k \models b$ and $w^{0,k-1}\mathsf{T}^\omega, L_0 \models P$. Here, $w^{0,-1}$ denotes the empty word.

— $w, L_0 \models \mathbf{not}\ P$ iff $\bar{w}, L_0 \not\models P$.

— $w, L_0 \models R$ iff there exist $0 \leq j < |w|$ and L_1 so that $w^{0,j}, L_0, L_1 \models R$.

— $w, L_0 \models (\ R\ |\text{->}\ P\ )$ iff for every $0 \leq j < |w|$ and L_1 so that $\bar{w}^{0,j}, L_0, L_1 \models R$, $w^{j..}, L_1 \models P$.

— $w, L_0 \models (\ P\ )$ iff $w, L_0 \models P$.

— $w, L_0 \models (\ P_1 \ \mathbf{or}\ P_2\ )$ iff $w, L_0 \models P_1$ or $w, L_0 \models P_2$.

— $w, L_0 \models (\ P_1 \ \mathbf{and}\ P_2\ )$ iff $w, L_0 \models P_1$ and $w, L_0 \models P_2$.

E.3.6.2 Weak and strong satisfaction by finite words

The definition is identical to that without local variables, but with the understanding that the underlying properties can have local variables.

E.4 Extended expressions

This subclause describes the semantics of several constructs that are used like expressions, but whose meaning at a point in a word can depend both on the letter at that point and on previous letters in the word. By abuse of notation, the meanings of these extended expressions are defined for letters denoted "w^j" even though they depend also on letters w^i for $i \leq j$. The reason for this abuse is to make clear the way these definitions should be used in combination with those in preceding subclauses.

E.4.1 Extended booleans

w denotes a nonempty finite or infinite word over Σ, j denotes an integer so that $0 \leq j < |w|$, and $T(V)$ denotes an instance of a clocked or unclocked sequence that is passed the local variables V as actual arguments.

— $w^j, L_0, L_1 \models T(V).\texttt{ended}$ iff there exist $0 \leq i \leq j$ and L so that both $w^{i,j}$, $\{\}$, $L \models T(V)$ and $L_1 = L_0|_D \cup L_V$, where $D = \text{dom}(L_0) - (\text{dom}(L) \cap V)$.

— $w^j, L_0, L_1 \models \texttt{@(}c\texttt{)} (T(V).\texttt{matched})$ iff there exists $0 \leq i < j$ so that $w^i, L_0, L_1 \models T(V).\texttt{ended}$ and $w^{i+1,j}$, $\{\}$, $\{\} \models \texttt{(!}c\texttt{ [*0:\$] ##1 }c\texttt{)}$.

— $w^j \models \texttt{@(}c\texttt{)\$stable(}e\texttt{)}$ iff there exists $0 \leq i < j$ so that $w^{i,j}$, $\{\}$, $\{\} \models \texttt{(}c\texttt{ ##1 }c\texttt{ [->1])}$ and $e[w^i] = e[w^j]$.

— $w^j \models \texttt{@(}c\texttt{)\$rose(}e\texttt{)}$ iff $b[w^j] = 1$ and (if there exists $0 \leq i < j$ so that $w^{i,j}$, $\{\}$, $\{\} \models \texttt{(}c\texttt{ ##1 }c\texttt{ [->1])}$ then $b[w^i] \neq 1$), where b is the LSB of e.

— $w^j \models \texttt{@(}c\texttt{)\$fell(}e\texttt{)}$ iff $b[w^j] = 0$ and (if there exists $0 \leq i < j$ so that $w^{i,j}$, $\{\}$, $\{\} \models \texttt{(}c\texttt{ ##1 }c\texttt{ [->1])}$ then $b[w^i] \neq 0$), where b is the LSB of e.

E.4.2 Past

w denotes a nonempty finite or infinite word over Σ, and j denotes an integer so that $0 \leq j < |w|$.

— Let $n \geq 1$. If there exist $0 \leq i < j$ so that $w^{i,j}$, $\{\}$, $\{\} \models \texttt{(}c\texttt{ ##1 }c\texttt{ [->}n-1\texttt{])}$, then $\texttt{@(}c\texttt{)\$past(}e\texttt{, }n\texttt{)}[w^j] = e[w^i]$. Otherwise, $\texttt{@(}c\texttt{)\$past(}e\texttt{, }n\texttt{)}[w^j]$ has the value $\texttt{x}$.

— $\texttt{\$past(}e\texttt{)} \equiv \texttt{\$past(}e\texttt{,1)}$.

E.5 Recursive properties

This subclause defines the neutral semantics of instances of recursive properties in terms of the neutral semantics of instances of nonrecursive properties. The latter can be expanded to properties in the abstract syntax by appropriate substitutions; therefore, their semantics is assumed to be understood.

Below are precise versions of the four restrictions given in and the precise definition of recursive property. The dependency digraph is the directed graph $\langle V, E \rangle$, where V is the set of all named properties and an order pair (p, q) is in E if, and only if, an instance of named property q appears in the declaration of named property p. For example, for the set of properties

```
property p1(v);
    v |=> p2(p3());
endproperty
```

```
property p2(v);
    a or (1'b1 |=> v);
endproperty

property p3;
    p1(a && b);
endproperty
```

the dependency digraph is

$$\langle \{p1, p2, p3\}, \{(p1,p2),(p1,p3),(p3,p1)\} \rangle$$

A named property is recursive if it is in a nontrivial, strongly connected component of the dependency digraph. An instance of named property q is recursive if it is in the declaration of a named property p so that p and q are in the same nontrivial, strongly connected component of the dependency digraph. Here, p and q need not be distinct properties. Define the weight of an instance of q in the declaration of p as the minimal number of time steps that are guaranteed from the beginning of the declaration of p until the instance of q. In the example above, the weights of `p2(p3())` and of `p3()` in `p1` are both one. Define the weight of an edge (p, q) in the dependency digraph as the minimal weight among the weights of instances of q in the declaration of p.

The following are the restrictions over recursive properties:
— RESTRICTION 1: The negation operator **not** cannot be applied to any property expression that instantiates a property from which a recursive property can be reached in the dependency digraph.
— RESTRICTION 2: The operator **disable iff** cannot be used in the declaration of a recursive property.
— RESTRICTION 3: In every cycle of the dependency digraph, the sum of the weights of the edges must be positive.
— RESTRICTION 4: For every recursive instance of `q` in the declaration of `p`, each actual argument expression `e` of the instance satisfies at least one of the following conditions:
    — `e` is itself a formal argument of `p`.
    — No formal argument of `p` appears in `e`.
    — `e` is passed to a formal argument of `q` that is typed and the set of values for the type is bounded.

Annex F

(normative)

DPI C layer

F.1 Overview

The SystemVerilog DPI allows direct inter-language function calls between SystemVerilog and any foreign programming language with a C function call protocol and linking model:

— Functions implemented in C and given import declarations in SystemVerilog can be called from SystemVerilog; such functions are referred to as *imported functions*.

— Functions implemented in SystemVerilog and specified in export declarations can be called from C; such functions are referred to as *exported functions*.

— Tasks implemented in SystemVerilog and specified in export declarations can be called from C; such functions are referred to as *exported tasks*.

— Functions implemented in C that can be called from SystemVerilog and can in turn call exported tasks; such functions are referred to as *imported tasks*.

The SystemVerilog DPI supports only SystemVerilog data types, which are the sole data types that can cross the boundary between SystemVerilog and a foreign language in either direction. On the other hand, the data types used in C code shall be C types; hence, the duality of types.

A value that is passed through the DPI is specified in SystemVerilog code as a value of SystemVerilog type, while the same value shall be specified in C code as a value of C type. Therefore, a pair of matching type definitions is required to pass a value through DPI: the SystemVerilog definition and the C definition.

It is the user's responsibility to provide these matching definitions. A tool (such as a SystemVerilog compiler) can facilitate this by generating C type definitions for the SystemVerilog definitions used in DPI for imported and exported functions.

Some SystemVerilog types are directly compatible with C types; defining a matching C type for them is straightforward. There are, however, SystemVerilog-specific types, namely packed types (arrays, structures, and unions), 2-state or 4-state, which have no natural correspondence in C. DPI defines a canonical representation of 4-state types that is exactly the same as the representation used by the VPI's avalue/bvalue representation of 4-state vectors. DPI defines a 2-state representation model that is consistent with the VPI 4-state model. DPI defines library functions to assist users in working with the canonical data representation.

The DPI C interface includes deprecated functions and definitions related to implementation-specific representation of packed array arguments. These functions are enabled by using the "DPI" specification string in import and export declarations (see 26.4). Refer to F.12 for details on the deprecated functionality.

Formal arguments in SystemVerilog can be specified as open arrays solely in import declarations; exported SystemVerilog functions cannot have formal arguments specified as open arrays. A formal argument is an open array when a range of one or more of its dimensions is unspecified (denoted in SystemVerilog by using empty square brackets, []). This corresponds to a relaxation of the DPI argument-matching rules (see 26.5.1). An actual argument shall match the corresponding formal argument regardless of the range(s) for its corresponding dimension(s), which facilitates writing generalized C code that can handle SystemVerilog arrays of different sizes.

The C layer of DPI basically uses normalized ranges. The term *normalized ranges* means [n-1:0] indexing for the packed part (packed arrays are restricted to one dimension) and [0:n-1] indexing for a dimension in the unpacked part of an array. Normalized ranges are used for the canonical representation of packed arrays in C and for SystemVerilog arrays passed as actual arguments to C, with the exception of actual arguments for open arrays. The elements of an open array can be accessed in C by using the same range of indices as defined in SystemVerilog for the actual argument for that open array and the same indexing as in SystemVerilog.

Function arguments are generally passed by some form of reference or by value. All formal arguments, except open arrays, are passed by direct reference or value, and, therefore, are directly accessible in C code. Only small values of SystemVerilog input arguments (see F.7.7) are passed by value. Formal arguments declared in SystemVerilog as open arrays are passed by a handle (type svOpenArrayHandle) and are accessible via library functions. Array-querying functions are provided for open arrays.

The C layer of DPI defines a portable binary interface. Once DPI C code is compiled into object code, the resulting object code shall work without recompilation in any compliant SystemVerilog implementation.

One normative include file, svdpi.h, is provided as part of the DPI C layer. This file defines all basic types, the canonical 2-state and 4-state data representation, and all interface functions.

F.2 Naming conventions

All names introduced by this interface shall conform to the following conventions:

— All names defined in this interface are prefixed with sv or SV_.

— Function and type names start with sv, followed by initially capitalized words with no separators, e.g., svLogicVecVal.

— Names of symbolic constants start with sv_, e.g., sv_x.

— Names of macro definitions start with SV_, followed by all uppercase words separated by a underscore (_), e.g., SV_GET_UNSIGNED_BITS.

F.3 Portability

DPI applications are always portable at the binary level. When compiled on a given platform, DPI object code shall work with every SystemVerilog simulator on that platform.

F.4 svdpi.h include file

The C layer of the DPI defines include file svdpi.h.

Applications that use the DPI with C code usually need this main include file. The include file svdpi.h defines the types for canonical representation of 2-state (bit) and 4-state (logic) values and passing references to SystemVerilog data objects. The file also provides function headers and defines a number of helper macros and constants.

The svdpi.h file is fully defined in Annex G. The content of svdpi.h does not depend on any particular implementation; all simulators shall use the same file. For more details on svdpi.h, see F.9.1 and Annex G.

This file may also contain the deprecated functions and data representations described in F.12. F.12 also describes the deprecated header svdpi_src.h, which defines the implementation-dependent representation of packed values.

F.5 Semantic constraints

NOTE—Constraints expressed here merely restate those expressed in 26.4.1.

Formal and actual arguments of both imported tasks or functions and exported tasks or functions are bound by the WYSIWYG principle: What You Specify Is What You Get. This principle is binding both for the caller and for the callee, in C code and in SystemVerilog code. For the callee, it guarantees the actual arguments are as specified for the formal ones. For the caller, it means the function call arguments shall conform with the types of the formal arguments, which might require type-coercion on the caller side.

Another way to state this is that no compiler (either C or SystemVerilog) can make argument coercions between a caller's declared formals and the callee's declared formals. This is because the callee's formal arguments are declared in a different language from the caller's formal arguments; hence there is no visible relationship between the two sets of formals. Users are expected to understand all argument relationships and provide properly matched types on both sides of the interface (see F.6.2).

In SystemVerilog code, the compiler can change the formal arguments of a native SystemVerilog task or function and modify its code accordingly because of optimizations, compiler pragmas, or command line switches. The situation is different for imported tasks and functions. A SystemVerilog compiler cannot modify the C code, perform any coercions, or make any changes whatsoever to the formal arguments of an imported task or function.

A SystemVerilog compiler shall provide any necessary coercions for the actual arguments of every imported task and function call. For example, a SystemVerilog compiler might truncate or extend bits of a packed array if the widths of the actual and formal arguments are different. Similarly, a C compiler can provide coercion for C types based on the relationship of the arguments in the exported task's and function's C prototype (formals) and the exported task's and function's C call site (actuals). However, a C compiler cannot provide such coercion for SystemVerilog types.

Thus, in each case of an inter-language function call, either C to SystemVerilog or SystemVerilog to C, the compilers expect, but cannot enforce, that the types on either side are compatible. It is, therefore, the user's responsibility to ensure that the imported/exported function types exactly match the types of the corresponding tasks or functions in the foreign language.

F.5.1 Types of formal arguments

The WYSIWYG principle guarantees the types of formal arguments of imported functions: an actual argument is guaranteed to be of the type specified for the formal argument, with the exception of open arrays (for which unspecified ranges are statically unknown). Formal arguments, other than open arrays, are fully defined by imported declaration; they shall have ranges of packed or unpacked arrays exactly as specified in the imported declaration. Only the SystemVerilog declaration site of the imported function is relevant for such formal arguments.

Formal arguments defined as open arrays have the size and ranges of the actual argument, i.e., have the ranges of packed or unpacked arrays exactly as that of the actual argument. The unsized ranges of open arrays are determined at a call site; the rest of the type information is specified at the import declaration. See also F.6.1.

Therefore, if a formal argument is declared as `bit [15:8] b []`, then the import declaration specifies that the formal argument is an unpacked array of packed bit array with bounds `15` to `8`, while the actual argument used at a particular call site defines the bounds for the unpacked part for that call.

F.5.2 Input arguments

Formal arguments specified in SystemVerilog as **input** must not be modified by the foreign language code. See also 26.4.1.2.

F.5.3 Output arguments

The initial values of formal arguments specified in SystemVerilog as **output** are undetermined and implementation dependent. See also 26.4.1.2.

F.5.4 Value changes for output and inout arguments

The SystemVerilog simulator is responsible for handling value changes for **output** and **inout** arguments. Such changes shall be detected and handled after the control returns from C code to SystemVerilog code.

F.5.5 Context and noncontext tasks and functions

Also refer to 26.4.3.

Some DPI imported tasks or functions or other interface functions called from them require that the context of their call be known. It takes special instrumentation of their call instances to provide such context; for example, a variable referring to the "current instance" might need to be set. To avoid any unnecessary overhead, imported tasks and function calls in SystemVerilog code are not instrumented unless the imported tasks or function is specified as context in its SystemVerilog import declaration.

The SystemVerilog context of DPI export tasks and functions must be known when they are called, including when they are called by imports. When an import invokes the `svSetScope` utility prior to calling the export, it sets the context explicitly. Otherwise, the context will be the context of the instantiated scope where the import declaration is located. Because imports with diverse instantiated scopes can export the same task or function, multiple instances of such an export can exist after elaboration. Prior to any invocations of `svSetScope`, these export instances would have different contexts, which would reflect their imported caller's instantiated scope.

For the sake of simulation performance, a noncontext imported task or function call shall not block SystemVerilog compiler optimizations. An imported task or function not specified as context shall not access any data objects from SystemVerilog other than its actual arguments. Only the actual arguments can be affected (read or written) by its call. Therefore, a call of noncontext imported task or function is not a barrier for optimizations. A context imported task or function, however, can access (read or write) any SystemVerilog data objects by calling PLI/VPI or by calling an embedded export task or function. Therefore, a call to a context task or function is a barrier for SystemVerilog compiler optimizations.

Only the calls of context imported tasks and functions are properly instrumented and cause conservative optimizations; therefore, only those tasks and functions can safely call all functions from other APIs, including PLI and VPI functions or exported SystemVerilog functions. For imported task or functions not specified as context, the effects of calling PLI, VPI, or SystemVerilog functions can be unpredictable; and such calls can crash if the callee requires a context that has not been properly set.

Special DPI utility functions exist that allow imported task and functions to retrieve and operate on their context. For example, the C implementation of an imported task or function can use `svGetScope()` to retrieve an `svScope` corresponding to the instance scope of its corresponding SystemVerilog import declaration. See F.8 for more details.

F.5.6 Pure functions

See also 26.4.2.

Only nonvoid functions with no **output** or **inout** arguments can be specified as **pure**. Functions specified as **pure** in their corresponding SystemVerilog import declarations shall have no side effects; their results need to depend solely on the values of their input arguments. Calls to such functions can be removed by SystemVerilog compiler optimizations or replaced with the values previously computed for the same values of the input arguments.

Specifically, a `pure` function is assumed not to directly or indirectly (i.e., by calling other functions) perform the following:

— Perform any file operations.

— Read or write anything in the broadest possible meaning, including input/output, environment variables, objects from the operating system or from the program or other processes, shared memory, sockets, etc.

— Access any persistent data, like global or static variables.

If a `pure` function does not obey the above restrictions, SystemVerilog compiler optimizations can lead to unexpected behavior, due to eliminated calls or incorrect results being used.

F.5.7 Memory management

See also 26.4.1.4.

The memory spaces owned and allocated by C code and SystemVerilog code are disjoined. Each side is responsible for its own allocated memory. Specifically, C code shall not free the memory allocated by SystemVerilog code (or the SystemVerilog compiler) nor expect SystemVerilog code to free the memory allocated by C code (or the C compiler). This does not exclude scenarios in which C code allocates a block of memory and then passes a handle (i.e., a pointer) to that block to SystemVerilog code, which in turn calls a C function that directly (if it is the standard function `free`) or indirectly frees that block.

NOTE—In this last scenario, a block of memory is allocated and freed in C code, even when the standard functions `malloc` and `free` are called directly from SystemVerilog code.

F.6 Data types

This subclause defines the data types of the C layer of the DPI.

F.6.1 Limitations

Packed arrays can have an arbitrary number of dimensions although they are eventually always equivalent to a one-dimensional packed array and treated as such. If the packed part of an array in the type of a formal argument in SystemVerilog is specified as multidimensional, the SystemVerilog compiler linearizes it. Although the original ranges are generally preserved for open arrays, if the actual argument has a

multidimensional packed part of the array, it shall be normalized into an equivalent one-dimensional packed array. (See F.6.5).

NOTE—The actual argument can have both packed and unpacked parts of an array; either can be multidimensional.

F.6.2 Duality of types: SystemVerilog types versus C types

A value that crosses the DPI is specified in SystemVerilog code as a value of SystemVerilog type, while the same value shall be specified in C code as a value of C type. Therefore, each data type that is passed through the DPI requires two matching type definitions: the SystemVerilog definition and C definition.

The user needs to provide such matching definitions. Specifically, for each SystemVerilog type used in the import declarations or export declarations in SystemVerilog code, the user shall provide the equivalent type definition in C reflecting the argument passing mode for the particular type of SystemVerilog value and the direction (**input**, **output**, or **inout**) of the formal SystemVerilog argument.

F.6.3 Data representation

DPI imposes the following additional restrictions on the representation of SystemVerilog data types:
— SystemVerilog types that are not packed and that do not contain packed elements have C-compatible representation.
— Basic integer and real data types are represented as defined in F.6.4.
— Enumeration types are represented by C base types that correspond to the enumeration types' SystemVerilog base types (see Table F-1). The base type determines whether an enumeration type is considered a small value (see 26.4.5). DPI supports all the SystemVerilog enumeration base types (see 4.10 and A.2.2.1). **integer** and **time** base types are represented as 4-state packed arrays in canonical form. Enumerated names are not available on the C side of the interface.
— Packed types are represented using the canonical format defined in F.6.7.
— Unpacked arrays embedded in a structure or union have C-compatible layout regardless of the type of elements. Similarly, standalone arrays passed as actuals to a sized formal argument have C-compatible representation.
— For a standalone array passed as an actual to an open array formal
    — If the element type is a 2- or 4-state scalar or packed type, then the representation is in canonical form.
    — Otherwise, the representation is C compatible. Therefore, an element of an array shall have the same representation as an individual value of the same type. Hence, an array's elements can be accessed from C code via normal C array indexing similarly to doing so for individual values.
— The natural order of elements for each dimension in the layout of an unpacked array shall be used, i.e., elements with lower indices go first. For SystemVerilog range [L:R], the element with SystemVerilog index min(L,R) has the C index 0 and the element with SystemVerilog index max(L,R) has the C index abs(L-R).

F.6.4 Basic types

Table F-1 defines the mapping between the basic SystemVerilog data types and the corresponding C types.

The DPI also supports the SystemVerilog and C unsigned integer data types that correspond to the mappings Table F-1 shows for their signed equivalents.

Table F-1—Mapping data types

SystemVerilog type	C type
byte	**char**
shortint	**short int**
int	**int**
longint	**long long**
real	**double**
shortreal	**float**
chandle	**void***
string	**const char***
bit[a]	**unsigned char**
logic[a]**/reg**	**unsigned char**

[a]Encodings for **bit** and **logic** are given in file `svdpi.h`. Reg
parameters can use the same encodings as logic parameters.

The **input** mode arguments of type **byte unsigned** and **shortint unsigned** are not equivalent to **bit**[7:0] or **bit**[15:0], respectively, because the former are passed as C types **unsigned char** and **unsigned short** and the latter are both passed by reference as `svBitVecVal` types. A similar lack of equivalence applies to passing such parameters by reference for **output** and **inout** modes, e.g., **byte unsigned** is passed as C type **unsigned char*** while **bit**[7:0] is passed by reference as `svBitVecVal`.

In addition to declaring DPI formal arguments of packed **bit** and **logic** arrays, it is also possible to declare formal arguments of packed **struct** and **union** types. DPI handles these types as if they were declared with equivalent one-dimensional packed array syntax. See 6.9.2.

Refer to F.6.8 for details on unpacked aggregate types that are composed of the basic types described in this subclause.

The handling of string types varies depending on the argument passing mode. Refer to F.7.10 for further details.

F.6.5 Normalized and linearized ranges

Packed arrays are treated as one-dimensional; the unpacked part of an array can have an arbitrary number of dimensions. Normalized ranges mean [n-1:0] indexing for the packed part and [0:n-1] indexing for a dimension of the unpacked part of an array. Normalized ranges are used for accessing all arguments but open arrays. The canonical representation of packed arrays also uses normalized ranges.

Linearizing a SystemVerilog array with multiple packed dimensions consists of treating an array with dimension sizes (i, j, k) as if it had a single dimension with size (i * j * k) and had been stored as a one-dimensional array. The one-dimensional array has the same layout as the corresponding multidimensional array stored in row-major order. User C code can take the original dimensions into account when referencing a linearized array element. For example, the bit in a SystemVerilog packed 2-state array with dimension sizes (i, j, k) and a SystemVerilog reference `myArray[l][m][n]` (where the ranges for l, m, and n have been normalized) maps to linearized C array **bit** (n + (m * k) + (l * j * k)).

F.6.6 Mapping between SystemVerilog ranges and C ranges

The SystemVerilog ranges for a formal argument specified as an open array are those of the actual argument for a particular call. Open arrays are accessible, however, by using their original ranges and the same indexing as in the SystemVerilog code.

For all other types of arguments, i.e., all arguments but open arrays, the SystemVerilog ranges are defined in the corresponding SystemVerilog import or export declaration. Normalized ranges are used for accessing such arguments in C code. C ranges for multiple packed dimensions are linearized. The mapping between SystemVerilog ranges and C ranges is defined as follows.

a) If a packed part of an array has more than one dimension, it is linearized as specified by the equivalence of packed types (see F.6.5 and 6.9.2).

b) A packed array of range `[L:R]` is normalized as `[abs(L-R):0]`; its MSB has a normalized index `abs(L-R)` and its LSB has a normalized index `0`.

c) The natural order of elements for each dimension in the layout of an unpacked array shall be used, i.e., elements with lower indices go first. For SystemVerilog range `[L:R]`, the element with SystemVerilog index `min(L,R)` has the C index `0` and the element with SystemVerilog index `max(L,R)` has the C index `abs(L-R)`.

The above range mapping from SystemVerilog to C applies to calls made in both directions, i.e., SystemVerilog calls to C and C calls to SystemVerilog.

For example, if **logic** `[2:3][1:3][2:0]` b `[1:10]` `[31:0]` is used in SystemVerilog, it needs to be defined in C as if it were declared in SystemVerilog in the following normalized form: **logic** `[17:0]` b `[0:9]` `[0:31]`.

F.6.7 Canonical representation of packed arrays

The DPI defines the canonical representation of packed 2-state (type `svBitVecVal`) and 4-state arrays (type `svBitVecVal`). `svLogicVecVal` is fully equivalent to type `s_vpi_vecval`, which is used to represent 4-state logic in VPI.

A packed array is represented as an array of one or more elements (of type `svBitVecVal` for 2-state values and `svLogicVecVal` for 4-state values), each element representing a group of 32 bits. The first element of an array contains the 32 LSBs, next element contains the 32 more significant bits, and so on. The last element can contain a number of unused bits. The contents of these unused bits are undetermined, and the user is responsible for the masking or the sign extension (depending on the sign) for the unused bits.

F.6.8 Unpacked aggregate arguments

Imported and exported DPI tasks and functions can make use of unpacked aggregate types as formal or actual arguments. Aggregate types include unpacked arrays, structures, and unions. Such types can be composed of packed elements, unpacked elements, or combinations of either kind of element, including subaggregates. Refer to Table F-1 for a list of legal basic types that can be used as nonaggregate elements in aggregate types. Also refer to 26.4.6.

In the case of an unpacked type that consists purely of unpacked elements (including subaggregates), the layout presented to the C programmer is guaranteed to be compatible with the C compiler's layout on the given operating system. It is also possible for unpacked aggregate types to include packed elements.

F.7 Argument passing modes

This subclause defines the ways to pass arguments in the C layer of the DPI.

F.7.1 Overview

Imported and exported function arguments are generally passed by some form of a reference, with the exception of small values of SystemVerilog input arguments (see F.7.7), which are passed by value. Similarly, the function result, which is restricted to small values, is passed by value, i.e., directly returned.

Formal arguments, except open arrays, are passed by direct reference or value and, therefore, are directly accessible in C code. Formal arguments declared in SystemVerilog as open arrays are passed by a handle (type `svOpenArrayHandle`) and are accessible via library functions.

F.7.2 Calling SystemVerilog tasks and functions from C

There is no difference in argument passing between calls from SystemVerilog to C and calls from C to SystemVerilog. Tasks and functions exported from SystemVerilog cannot have open arrays as arguments. Apart from this restriction, the same types of formal arguments can be declared in SystemVerilog for exported tasks and functions and imported tasks and functions. A task or function exported from SystemVerilog shall have the same function header in C as would an imported function with the same function result type and same formal argument list. In the case of arguments passed by reference, an actual argument to SystemVerilog task and function called from C shall be allocated using the same layout of data as SystemVerilog uses for that type of argument; the caller is responsible for the allocation. It can be done while preserving the binary compatibility (see F.11.5 and F.12).

Calling a SystemVerilog task from C is the same as calling a SystemVerilog function from C with the exception that the return type of an exported task is an **int** value that has a special meaning related to **disable** statements. See 26.8 for details on disable processing by DPI imported tasks and functions.

F.7.3 Argument passing by value

Only small values of formal input arguments (see F.7.7) are passed by value. Function results are also directly passed by value. The user needs to provide the C type equivalent to the SystemVerilog type of a formal argument if an argument is passed by value.

F.7.4 Argument passing by reference

For arguments passed by reference, a reference (a pointer) to the actual data object is passed. In the case of packed data, a reference to a canonical data object is passed. The actual argument is usually allocated by a caller. The caller can also pass a reference to an object already allocated somewhere else, for example, its own formal argument passed by reference.

If an argument of type `T` is passed by reference, the formal argument shall be of type `T*`. Packed arrays are passed using a pointer to the appropriate canonical type definition, either `svLogicVecVal*` or `svBitVecVal*`.

F.7.5 Allocating actual arguments for SystemVerilog-specific types

This is relevant only for calling exported SystemVerilog tasks or functions from C code. The caller is responsible for allocating any actual arguments that are passed by reference.

Static allocation requires knowledge of the relevant data type. If such a type involves SystemVerilog packed arrays, corresponding C arrays of canonical data types (either `svLogicVecVal` or `svBitVecVal`) must be allocated and initialized before being passed by reference to the exported SystemVerilog task or function.

F.7.6 Argument passing by handle—open arrays

Arguments specified as open (unsized) arrays are always passed by a handle, regardless of direction of the SystemVerilog formal argument, and are accessible via library functions. The actual implementation of a handle is simulator-specific and transparent to the user. A handle is represented by the generic pointer `void *` (`typedef`ed to `svOpenArrayHandle`). Arguments passed by handle shall always have a **const** qualifier because the user shall not modify the contents of a handle.

F.7.7 Input arguments

input arguments of imported functions implemented in C shall always have a **const** qualifier.

input arguments, with the exception of open arrays, are passed by value or by reference, depending on the size. Small values of formal input arguments are passed by value. The following data types are considered small:

— **byte**, **shortint**, **int**, **longint**, **real**, **shortreal**
— Scalar **bit** and **logic**
— **chandle**, **string**

input arguments of other types are passed by reference.

F.7.8 Inout and output arguments

inout and **output** arguments, with the exception of open arrays, are always passed by reference. Specifically, packed arrays are passed, accordingly, as `svBitVecVal*` or `svLogicVecVal*`. The same rules about unused bits apply as in F.6.7.

F.7.9 Function result

Types of a function result are restricted to the following SystemVerilog data types (see Table F-1 for the corresponding C type):

— **byte**, **shortint**, **int**, **longint**, **real**, **shortreal**, **chandle**, **string**
— Scalar values of type **bit** and **logic**

Encodings for **bit** and **logic** are given in file `svdpi.h`. Refer to F.9.1.1.

F.7.10 String arguments

The layout of SystemVerilog string objects is implementation dependent. However, when a string value is passed from SystemVerilog to C, implementations shall ensure that all characters in the string are laid out in memory per C string conventions, including a trailing null character present at the end of the C string. Similarly, users shall ensure that any C strings passed to SystemVerilog are properly null-terminated.

The direction mode for string arguments applies to the pointer to the string (i.e., the `const char*` variable in Table F-1), not to the characters that compose the string.

Thus, the direction modes have the following meanings for imported tasks and functions:

— An **input** mode string is accessed through a pointer value that is provided by SystemVerilog and that the user shall not free. No user changes to this pointer value are propagated back to the SystemVerilog sphere.

— An **output** mode string does not arrive at the C interface with a meaningful value. It is represented by a const char** variable. Upon return to SystemVerilog, the user shall have written a valid and initialized const char* address into the const char** variable. SystemVerilog shall not free memory accessed through this address.

— An **inout** mode string arrives at the C interface with a valid string address value stored in a const char** variable. The user shall not free the string's storage. Any user changes to the string shall be effected by the user supplying a new pointer value, which points to new string contents and which SystemVerilog shall not attempt to free. The user provides a new string pointer value by writing the string's address into the const char** variable. If the user does so, SystemVerilog copies the indicated string contents into its memory space and undertakes any actions sensitive to this change.

The direction modes have the following meanings for exported tasks and functions:

— An **input** mode string is passed to SystemVerilog through a const char* pointer. SystemVerilog only reads from the string. It shall not modify the characters that compose the string.

— An **output** mode string is represented by a const char** variable. No meaningful initial value is stored in the pointer variable. SystemVerilog shall write a valid string address into the output const char** variable. The user shall not make any assumptions about the lifetime of the output string's storage, and the C code shall not free the string memory. If it is desired to refer to the string's value at some point in the future, the user shall copy the string value to memory owned by the C domain.

— An **inout** mode string is represented by a const char** variable that contains a pointer to memory allocated and initialized by the user. SystemVerilog only reads from the user's string storage, and it will not attempt to modify or free this storage. If SystemVerilog needs to effect a change in the value of the inout mode string, then a valid SystemVerilog string address is written into the const char** variable. The user shall not make any assumptions about the lifetime of this string storage, nor should the SystemVerilog storage be freed by C code. If it is desired to refer to the modified string value at some point in the future, the user shall copy the string value to memory owned by the C domain.

F.8 Context tasks and functions

Some DPI imported tasks and functions require that the context of their call be known. For example, those calls can be associated with instances of C models that have a one-to-one correspondence with instances of SystemVerilog modules that are making the calls. Alternatively, a DPI imported task or function might need to access or modify simulator data structures using PLI or VPI calls or by making a call back into System-Verilog via an export task or function. Context knowledge is required for such calls to function properly. It can take special instrumentation of their call to provide such context.

To avoid any unnecessary overhead, imported task and function calls in SystemVerilog code are not instrumented unless the imported task or function is specified as context in its SystemVerilog import declaration. A small set of DPI utility functions are available to assist programmers when working with context tasks or functions (see F.8.3). If those utility functions are used with any noncontext function, a system error shall result.

F.8.1 Overview of DPI and VPI context

Both DPI task and functions and VPI/PLI functions might need to understand their context. However, the meaning of the term is different for the two categories of task and functions.

DPI imported tasks and functions are essentially proxies for native SystemVerilog tasks and functions. Native SystemVerilog tasks and functions always operate in the scope of their declaration site. For example, a native SystemVerilog function `f()` can be declared in a module `m`, which is instantiated as `top.i1_m`. The `top.i1_m` instance of `f()` can be called via hierarchical reference from code in a distant design region. Function `f()` is said to execute in the context (i.e., instantiated scope) of `top.i1_m` because it has unqualified visibility only for variables local to that specific instance of `m`. Function `f()` does not have unqualified visibility for any variables in the calling code's scope.

DPI imported tasks and functions follow the same model as native SystemVerilog tasks and functions. They execute in the context of their surrounding declarative scope, rather than the context of their call sites. This type of context is termed *DPI context*.

This is in contrast to VPI and PLI functions. Such functions execute in a context associated with their call sites. The VPI or PLI programming model relies on C code's ability to retrieve a context handle associated with the associated system task's call site and then to work with the context handle to glean information about arguments, items in the call site's surrounding declarative scope, etc. This type of context is termed *VPI context*.

The SystemVerilog context of DPI export tasks and functions must be known when they are called, including when they are called by imports. When an import invokes the `svSetScope` utility prior to calling the export, it sets the context explicitly. Otherwise, the context will be the context of the instantiated scope where the import declaration is located. Because imports with diverse instantiated scopes can export the same task or function, multiple instances of such an export can exist after elaboration. Prior to any invocations of `svSetScope`, these export instances would have different contexts, which would reflect their imported caller's instantiated scope.

F.8.2 Context of imported and export tasks and functions

DPI imported and export tasks and functions can be declared anywhere a normal SystemVerilog task or function can be declared. Specifically, they can be declared in **module**, **program**, **interface**, or **generate** declarative scope.

A context imported task or function executes in the context of the instantiated scope surrounding its declaration. In other words, such tasks and functions can see other variables in that scope without qualification. As explained in F.8.1, this should not be confused with the context of the task's or function's call site, which can actually be anywhere in the SystemVerilog design hierarchy. The context of an imported or exported task or function corresponds to the fully qualified name of the task or function, minus the task or function name itself.

The context property is transitive through imported and export context tasks and functions declared in the same scope. In other words, if an imported task or function is running in a certain context and if it in turn calls an exported task or function that is available in the same context, the exported task or function can be called without any use of `svSetScope()`. For example, consider a SystemVerilog call to a native function `f()`, which in turn calls a native function `g()`. Now replace the native function `f()` with an equivalent imported context C function, `f'()`. The system shall behave identically regardless if `f()` or `f'()` is in the call chain above `g()`. `g()` has the proper execution context in both cases.

F.8.3 Working with DPI context tasks and functions in C code

DPI defines a small set of functions to help programmers work with DPI context tasks and functions. The term *scope* is used in the task or function names for consistency with other SystemVerilog terminology. The terms *scope* and *context* are equivalent for DPI tasks and functions.

There are functions that allow the user to retrieve and manipulate the current operational scope. It is an error to use these functions with any C code that is not executing under a call to a DPI context imported task or function.

There are also functions that provide users with the power to set data specific to C models into the System-Verilog simulator for later retrieval. These are the "put" and "get" user data functions, which are similar to facilities provided in VPI and PLI.

The put and get user data functions are flexible and allow for a number of use models. Users might wish to share user data across multiple context imported functions defined in the same SystemVerilog scope. Users might wish to have unique data storage on a per-function basis. Shared or unique data storage is controllable by a user-defined key.

To achieve shared data storage, a related set of context imported tasks and functions should all use the same user key. To achieve unique data storage, a context import task or function should use a unique key, and it is a requirement on the user that such a key be truly unique from all other keys that could possibly be used by C code. This includes completely unknown C code that could be running in the same simulation. It is suggested that taking addresses of static C symbols (such as a function pointer or an address of some static C data) always be done for user key generation. Generating keys based on arbitrary integers is not a safe practice.

It is never possible to share user data storage across different contexts. For example, if a Verilog module `m` declares a context imported task or function `f`, and `m` is instantiated more than once in the SystemVerilog design, then `f` shall execute under different values of `svScope`. No such executing instances of `f` can share user data with each other, at least not using the system-provided user data storage area accessible via `svPutUserData()`.

A user wanting to share a data area across multiple contexts must do so by allocating the common data area and then storing the pointer to it individually for each of the contexts in question via multiple calls to `svPutUserData()`. This is because, although a common user key can be used, the data must be associated with the individual scopes (denoted by `svScope`) of those contexts.

```
    /* Functions for working with DPI context functions */

    /* Retrieve the active instance scope currently associated with the executing
     * imported function.
     * Unless a prior call to svSetScope has occurred, this is the scope of the
     * function's declaration site, not call site.
     * The return value is undefined if this function is invoked from a
    noncontext
     * imported function.
     */
    svScope svGetScope();

    /* Set context for subsequent export function execution.
     * This function must be called before calling an export function, unless
     * the export function is called while executing an extern function. In that
     * case the export function shall inherit the scope of the surrounding extern
     * function. This is known as the "default scope".
```

```
 * The return is the previous active scope (per svGetScope)
 */
svScope svSetScope(const svScope scope);

/* Gets the fully qualified name of a scope handle */
const char* svGetNameFromScope(const svScope);

/* Retrieve svScope to instance scope of an arbitrary function declaration.
 * (can be either module, program, interface, or generate scope)
 * The return value shall be NULL for unrecognized scope names.
 */
svScope svGetScopeFromName(const char* scopeName);

/* Store an arbitrary user data pointer for later retrieval by
svGetUserData()
 * The userKey is generated by the user. It must be guaranteed by the user to
 * be unique from all other userKey's for all unique data storage
requirements
 * It is recommended that the address of static functions or variables in the
 * user's C code be used as the userKey.
 * It is illegal to pass in NULL values for either the scope or userData
 * arguments. It is also an error to call svPutUserData() with an invalid
 * svScope. This function returns -1 for all error cases, 0 upon success. It
is
 * suggested that userData values of 0 (NULL) not be used as otherwise it can
 * be impossible to discern error status returns when calling svGetUserData()
 */
int svPutUserData(const svScope scope, void *userKey, void* userData);

/* Retrieve an arbitrary user data pointer that was previously
 * stored by a call to svPutUserData(). See the comment above
 * svPutUserData() for an explanation of userKey, as well as
 * restrictions on NULL and illegal svScope and userKey values.
 * This function returns NULL for all error cases, and a non-Null
 * user data pointer upon success.
 * This function also returns NULL in the event that a prior call
 * to svPutUserData() was never made.
 */
void* svGetUserData(const svScope scope, void* userKey);

/* Returns the file and line number in the SV code from which the extern call
 * was made. If this information available, returns TRUE and updates fileName
 * and lineNumber to the appropriate values. Behavior is unpredictable if
 * fileName or lineNumber are not appropriate pointers. If this information
is  * not available return FALSE and contents of fileName and lineNumber not
 * modified. Whether this information is available or not is implementation-
 * specific. Note that the string provided (if any) is owned by the SV
 * implementation and is valid only until the next call to any SV function.
 * Applications must not modify this string or free it
 */
int svGetCallerInfo(char **fileName, int *lineNumber);
```

F.8.4 Example 1—Using DPI context functions

```
SV Side:
     // Declare an imported context sensitive C function with cname
"MyCFunc"
     import "DPI-C" context MyCFunc = function integer MapID(int portID);
```

```
C Side:
        // Define the function and model class on the C++ side:
        class MyCModel {
        private:
           int locallyMapped(int portID); // Does something interesting...
        public:
           // Constructor
           MyCModel(const char* instancePath) {
              svScope svScope = svGetScopeByName(instancePath);

              // Associate "this" with the corresponding SystemVerilog scope
              // for fast retrieval during run time.
              svPutUserData(svScope, (void*) MyCFunc, this);
           }

        friend int MyCFunc(int portID);
        };

        // Implementation of imported context function callable in SV
        int MyCFunc(int portID) {
           // Retrieve SV instance scope (i.e., this function's context).
           svScope = svGetScope();

           // Retrieve and make use of user data stored in SV scope
           MyCModel* me = (MyCModel*)svGetUserData(svScope, (void*) MyCFunc);
           return me->locallyMapped(portID);
        }
```

F.8.5 Relationship between DPI and either VPI or PLI

DPI allows C code to run in the context of a SystemVerilog simulation; thus it is natural for users to consider using VPI or PLI C code from within imported tasks and functions.

There is no specific relationship defined between DPI and the existing VPI and PLI. Programmers must make no assumptions about how DPI and the other interfaces interact. For example, a vpiHandle is not equivalent to an svOpenArrayHandle, and the two must not be interchanged and passed between functions defined in two different interface standards.

If a user wants to call VPI or PLI functions from within an imported task or function, the imported task or function must be flagged with the context qualifier.

Not all VPI or PLI functionality is available from within DPI context imported tasks and functions. For example, a SystemVerilog imported task or function is not a system task, and thus making the following call from within an imported task or function would result in an error:

```
/* Get handle to system task call site in preparation for argument scan */
vpiHandle myHandle = vpi_handle(vpiSysTfCall, NULL);
```

Similarly, the receipt of misctf callbacks and other activities associated with system tasks are not supported inside DPI imported tasks and functions. Users should use VPI or PLI if they wish to accomplish such actions.

However, the following kind of code is guaranteed to work from within DPI context imported tasks and functions:

```
    /* Prepare to scan all top-level modules */
    vpiHandle myHandle = vpi_iterate(vpiModule, NULL);
```

F.9 Include files

The C layer of the DPI defines one include file, `svdpi.h`. This file is implementation independent and defines the canonical representation, all basic types, and all interface functions. The actual file is shown in Annex G.

F.9.1 Include file svdpi.h

Applications that use the DPI with C code usually need this main include file. The include file `svdpi.h` defines the types for canonical representation of 2-state (**bit**) and 4-state (**logic**) values and passing references to SystemVerilog data objects, provides function headers, and defines a number of helper macros and constants.

This standard fully defines the `svdpi.h` file. The content of `svdpi.h` does not depend on any particular implementation or platform; all simulators shall use the same file. Subclauses F.9.1.1, F.9.1.2, and F.9.1.3 (and F.12) detail the contents of the `svdpi.h` file.

F.9.1.1 Scalars of type bit and logic

```
    /* canonical representation */

    #define sv_0   0
    #define sv_1   1
    #define sv_z   2  /* representation of 4-st scalar z */
    #define sv_x   3  /* representation of 4-st scalar x */

    /* common type for 'bit' and 'logic' scalars. */
    typedef unsigned char svScalar;

    typedef svScalar svBit;    /* scalar */
    typedef svScalar svLogic;  /* scalar */
```

F.9.1.2 Canonical representation of packed arrays

```
    /*
     * DPI representation of packed arrays.
     * 2-state and 4-state vectors, exactly the same as PLI's avalue/bvalue.
     */
    #ifndef VPI_VECVAL
    #define VPI_VECVAL
    typedef struct vpi_vecval {
       uint32_t a;
       uint32_t b;
    } s_vpi_vecval, *p_vpi_vecval;
    #endif

    /* (a chunk of) packed logic array */
    typedef s_vpi_vecval svLogicVecVal;

    /* (a chunk of) packed bit array */
    typedef uint32_t svBitVecVal;
```

```
/* Number of chunks required to represent the given width packed array */
#define SV_PACKED_DATA_NELEMS(WIDTH) (((WIDTH) + 31) >> 5)

/*
 * Because the contents of the unused bits is undetermined,
 * the following macros can be handy.
 */
#define SV_MASK(N) (~(-1 << (N)))

#define SV_GET_UNSIGNED_BITS(VALUE, N) \
    ((N) == 32 ? (VALUE) : ((VALUE) & SV_MASK(N)))

#define SV_GET_SIGNED_BITS(VALUE, N) \
    ((N) == 32 ? (VALUE) : \
    (((VALUE) & (1 << (N))) ? ((VALUE) | ~SV_MASK(N)) : ((VALUE) &
SV_MASK(N))))
```

F.9.1.3 Implementation-dependent representation

The svDpiVersion() function returns a string indicating which DPI standard is supported by the simulator and in particular which canonical value representation is being provided. Simulators implementing the current standard, i.e., the VPI-based canonical value, must return the string "P1800-2005". Simulators implementing to the prior Accellera SV3.1a standards, and thus using the svLogicVec32 value representation, shall return the string "SV3.1a".

```
/* Returns either version string "P1800-2005" or "SV3.1a" */
const char* svDpiVersion();

/* a handle to a scope (an instance of a module or an interface) */
typedef void *svScope;

/* a handle to a generic object (actually, unsized array) */
typedef void* svOpenArrayHandle;
```

F.9.2 Example 2—Simple packed array application

SystemVerilog:

```
typedef struct {int x; int y;} pair;
import "DPI-C" function void foo(input int i1, pair i2,
                                 output logic [63:0] o3);

export "DPI-C" function exported_sv_func;

function void exported_sv_func(input int i, output int o [0:7]);
   begin ... end
endfunction
```

C:

```
#include "svdpi.h"

typedef struct {int x; int y;} pair;

extern void exported_sv_func(int, int *); /* imported from SystemVerilog */

void foo(const int i1, const pair *i2, svLogicVecVal* o3)
```

```
{
   int tab[8];

   printf("%d\n", i1);
   o3[0].a = i2->x;
   o3[0].b = 0;
   o3[1].a = i2->y;
   o3[1].b = 0;

   /* call SystemVerilog */
   exported_sv_func(i1, tab); /* tab passed by reference */
   ...
}
```

F.9.3 Example 3—Application with complex mix of types

SystemVerilog:

```
typedef struct {int a; bit [6:1][1:8] b [65:2]; int c;} triple;
   // troublesome mix of C types and packed arrays
import "DPI-C" function void foo(input triple t);

export "DPI-C" function exported_sv_func;

function void exported_sv_func(input int i, output logic [63:0] o);
   begin ... end
endfunction
```

C:

```
#include "svdpi.h"
typedef struct {
   int a;
   svBitVecVal b[64][SV_PACKED_DATA_NELEMS(6*8)];
   int c;
} triple;

/* Note that 'b' is defined as for 'bit [6*8-1:0] b [63:0]' */

extern void exported_sv_func(int, svLogicVecVal*); /* imported from
                                                 SystemVerilog */

void foo(const triple *t)
{
   int i;
   svBitVecVal   aB;
   svLogicVecVal aL[SV_PACKED_DATA_NELEMS(64)];

   /* aB holds results of part-select from packed bit array 'b' in
      struct triple. */
   /* aL holds the packed logic array filled in by the export function. */

   printf("%d %d\n", t->a, t->c);
   for (i = 0; i < 64; i++) {
      /* Read least significant byte of each word of b into aB, then
process...*/
      svGetPartSelBit(&aB, t->b[i], 0, 8);
```

```
      ...
    }
    ...
    /* Call SystemVerilog */
    exported_sv_func(2, aL); /* Export function writes data into
                                output arg "aL" */
    ...
}
```

F.10 Arrays

Normalized ranges are used for accessing SystemVerilog arrays, with the exception of formal arguments specified as open arrays.

F.10.1 Example 4—Using packed 2-state arguments

This example shows two alternatives for working with 2-state packed data types. The first argument shows classical **int**-to-**int** correspondence per Table F-1. The second argument demonstrates that a DPI formal argument can be of a C-compatible type and that arbitrary 2-state bit vector actual arguments can be associated with that C-compatible formal argument. The third argument shows a portable technique for handling an arbitrary width 2-state vector. This technique is less efficient than techniques involving C-compatible formal arguments, but it is required when 2-state vectors exceed 64 bits in length.

```
// SV code
module m;

   parameter W = 33;
   int abv1;
   bit [29:0]  abv2;
   bit [W-1:0] abv3;

   // Two ways of handling 2-state packed array arguments
   import "DPI-C" function void foo7(input int unsigned fbv1,
                                     input int unsigned fbv2,
                                     input bit [W-1:0] fbv3);

   initial
      foo7(abv1, abv2, abv3);
   endmodule


/* C code */
void foo7(unsigned int fbv1, unsigned int fbv2,
                 const svBitVecVal* fbv3)
{
   printf("fbv1 is %d, fbv2 is %d\n", fbv1, fbv2);
   /* Use of the 2-state svdpi utilities is needed to transform fbv3 into a
      C representation */
}
```

F.10.2 Multidimensional arrays

Multiple packed dimensions of a SystemVerilog array are linearized (see F.6.5). Unpacked arrays can have an arbitrary number of dimensions.

F.10.3 Example 5—Using packed struct and union arguments

This example shows how packed **struct** and **union** arguments correspond to one-dimensional packed array arguments.

```
// SV code
module m;

    typedef bit [2:0] A;
    typedef struct packed { bit a; bit b; bit c; } S;
    typedef union packed { A a; S s; } U;
    S s;
    U u;
    A a;

    // Import function takes three arguments
    import "DPI-C" function void foo8(input A fa, input S fs, input U fu);

    initial begin
       s.a = 1'b1;
       s.b = 1'b0;
       s.c = 1'b0;
       a = 3'b100;
       u.a = 3'b100;
       foo8(a, s, u);
    end

endmodule

/* C code */
void foo8(
    const svBitVecVal* fa,
    const svBitVecVal* fs,
    const svBitVecVal* fu)
{
     printf("fa is %d, fs is %d, fu is %d\n", *fa, *fs, *fu);
}
```

The output of the printf will be "fa is 4, fs is 4, fu is 4".

F.10.4 Direct access to unpacked arrays

Unpacked arrays, with the exception of formal arguments specified as open arrays, shall have the same layout as used by a C compiler; they are accessed using C indexing (see F.6.6).

F.10.5 Utility functions for working with the canonical representation

Packed arrays are accessible via canonical representation. This C layer interface provides utility functions for working with bit-selects and limited (up to 32-bit) part-selects in the canonical representation.

A part-select is a slice of a packed array of types **bit** or **logic**. Array slices are not supported for unpacked arrays. Functions for part-selects only allow access (read/write) to a narrow subrange of up to 32 bits. If the specified range of a part-select is not fully contained within the normalized range of an array, the behavior is undetermined.

DPI utilities behave in the following way, given part-select arguments of width `w` and starting index `i`: A utility puts part-select source bits `[w-1:0]` into destination bits `[(i+w-1):i]` without changing the values of destination bits that surround the part-select. A utility gets part-select source bits `[(i+w-1):i]` and copies them into destination bits `[w-1:0]`. If $w < 32$, destination bits `[31:w]` shall be left unchanged by the get part-select operation.

```
/*
 * Bit-select utility functions.
 *
 * Packed arrays are assumed to be indexed n-1:0,
 * where 0 is the index of LSB
 */

/* s=source, i=bit-index */
svBit svGetBitselBit(const svBitVecVal* s, int i);
svLogic svGetBitselLogic(const svLogicVecVal* s, int i);

/* d=destination, i=bit-index, s=scalar */
void svPutBitselBit(svBitVecVal* d, int i, svBit s);
void svPutBitselLogic(svLogicVecVal* d, int i, svLogic s);

/*
 * Part-select utility functions.
 *
 * A narrow (<=32 bits) part-select is extracted from the
 * source representation and written into the destination word.
 *
 * Normalized ranges and indexing [n-1:0] are used for both arrays.
 *
 * s=source, d=destination, i=starting bit index, w=width
 * like for variable part-selects; limitations: w <= 32
 */
void svGetPartselBit(svBitVecVal* d, const svBitVecVal* s, int i, int w);
void svGetPartselLogic(svLogicVecVal* d, const svLogicVecVal* s, int i, int
w);

void svPutPartselBit(svBitVecVal* d, const svBitVecVal s, int i, int w);
void svPutPartselLogic(svLogicVecVal* d, const svLogicVecVal s, int i, int
w);
```

F.11 Open arrays

Formal arguments specified as open arrays allows passing actual arguments of different sizes (i.e., different range and/or different number of elements), which facilitates writing more general C code that can handle SystemVerilog arrays of different sizes. The elements of an open array can be accessed in C by using the same range of indices and the same indexing as in SystemVerilog. Plus, inquiries about the dimensions and the original boundaries of SystemVerilog actual arguments are supported for open arrays.

NOTE—Both packed and unpacked array dimensions can be unsized.

All formal arguments declared in SystemVerilog as open arrays are passed by handle (type `svOpenArray-Handle`), regardless of the direction of a SystemVerilog formal argument. Such arguments are accessible via interface functions.

F.11.1 Actual ranges

The formal arguments defined as open arrays have the size and ranges of the actual argument, as determined on a per-call basis. The programmer shall always have a choice about whether to specify a formal argument as a sized array or as an open (unsized) array.

In the former case, all indices are normalized on the C side (i.e., `0` and up), and the programmer needs to know the size of an array and be capable of determining how the ranges of the actual argument map onto C-style ranges (see F.6.6).

   Tip: Programmers can decide to use `[n:0]name[0:k]` style ranges in SystemVerilog.

In the latter case, i.e., an open array, individual elements of a packed array are accessible via interface functions, which facilitate the SystemVerilog style of indexing with the original boundaries of the actual argument.

If a formal argument is specified as a sized array, then it shall be passed by reference, with no overhead, and is directly accessible as a normalized array. If a formal argument is specified as an open (unsized) array, then it shall be passed by handle, with some overhead, and is mostly indirectly accessible, again with some overhead, although it retains the original argument boundaries.

NOTE—This provides some degree of flexibility and allows the programmer to control the trade-off of performance versus convenience.

The following example shows the use of sized versus unsized arrays in SystemVerilog code:

```
// both unpacked arrays are 64 by 8 elements, packed 16-bit each
logic [15: 0] a_64x8 [63:0][7:0];
logic [31:16] b_64x8 [64:1][-1:-8];

import "DPI-C" function void foo(input logic [] i [][]);
      // 2-dimensional unsized unpacked array of unsized packed logic

import "DPI-C" function void boo(input logic [31:16] i [64:1][-1:-8]);
      // 2-dimensional sized unpacked array of sized packed logic

foo(a_64x8);
foo(b_64x8); // C code can use original ranges [31:16][64:1][-1:-8]

boo(b_64x8); // C code must use normalized ranges [15:0][0:63][0:7]
```

F.11.2 Array querying functions

These functions are modeled upon the SystemVerilog array querying functions and use the same semantics (see 22.6).

If the dimension is `0`, then the query refers to the packed part (which is one-dimensional) of an array, and dimensions `>` `0` refer to the unpacked part of an array.

```
/* h= handle to open array, d=dimension */
int svLeft(const svOpenArrayHandle h, int d);
int svRight(const svOpenArrayHandle h, int d);
int svLow(const svOpenArrayHandle h, int d);
int svHigh(const svOpenArrayHandle h, int d);
int svIncrement(const svOpenArrayHandle h, int d);
int svSize(const svOpenArrayHandle h, int d);
```

```
int svDimensions(const svOpenArrayHandle h);
```

F.11.3 Access functions

There are library functions available for copying data between open array handles and canonical form buffers provided by the C programmer. Likewise, there are functions to obtain the actual address of System-Verilog data objects or of an individual element of an unpacked array.

Depending on the type of an element of an unpacked array, different access methods shall be used when working with elements.

— Packed arrays (`bit` or `logic`) are accessed via copying to or from the canonical representation.

— Scalars (1-bit value of type `bit` or `logic`) are accessed (read or written) directly.

— Other types of values (e.g., structures) are accessed via generic pointers; a library function calculates an address, and the user needs to provide the appropriate casting.

— Scalars and packed arrays are accessible via pointers only if the implementation supports this functionality (per array), e.g., one array can be represented in a form that allows such access, while another array might use a compacted representation that renders this functionality unfeasible (both occurring within the same simulator).

SystemVerilog allows arbitrary dimensions and, hence, an arbitrary number of indices. To facilitate this, variable argument list functions shall be used. For the sake of performance, specialized versions of all indexing functions are provided for one, two, or three indices.

F.11.4 Access to actual representation

The following functions provide an actual address of the whole array or of its individual elements. These functions shall be used for accessing elements of arrays of types compatible with C. These functions are also useful for vendors because they provide access to the actual representation for all types of arrays.

If the actual layout of the SystemVerilog array passed as an argument for an open unpacked array is different from the C layout, then it is not possible to access such an array as a whole; therefore, the address and size of such an array shall be undefined (0, to be exact). Nonetheless, the addresses of individual elements of an array shall be always supported.

NOTE—No specific representation of an array is assumed here; hence, all functions use a generic pointer `void *`.

```
/* a pointer to the actual representation of the whole array of any type */
/* NULL if not in C layout */
void *svGetArrayPtr(const svOpenArrayHandle);

int svSizeOfArray(const svOpenArrayHandle); /* total size in bytes or 0 if
not
                                            in C layout */

/* Return a pointer to an element of the array
   or NULL if index outside the range or null pointer */

void *svGetArrElemPtr(const svOpenArrayHandle, int indx1, ...);

/* specialized versions for 1-, 2- and 3-dimensional arrays: */
void *svGetArrElemPtr1(const svOpenArrayHandle, int indx1);
void *svGetArrElemPtr2(const svOpenArrayHandle, int indx1, int indx2);
void *svGetArrElemPtr3(const svOpenArrayHandle, int indx1, int indx2,
```

```
        int indx3);
```

Access to an individual array element via pointer makes sense only if the representation of such an element is the same as it would be for an individual value of the same type. Representation of array elements of type `scalar` or *packed value* is implementation dependent; the above functions shall return NULL if the representation of the array elements differs from the representation of individual values of the same type.

F.11.5 Access to elements via canonical representation

This group of functions is meant for accessing elements that are packed arrays (`bit` or `logic`).

The following functions copy a single vector from a canonical representation to an element of an open array or copy the other way around. The element of an array is identified by indices, bound by the ranges of the actual argument, i.e., the original SystemVerilog ranges are used for indexing.

```
    /* functions for translation between simulator and canonical representations*/
    /* s=source, d=destination */

    /* From user space into simulator storage */
    void svPutBitArrElemVecVal(const svOpenArrayHandle d, const svBitVecVal* s,
            int indx1, ...);
    void svPutBitArrElem1VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
            int indx1);
    void svPutBitArrElem2VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
            int indx1, int indx2);
    void svPutBitArrElem3VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
            int indx1, int indx2, int indx3);
    void svPutLogicArrElemVecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
            int indx1, ...);
    void svPutLogicArrElem1VecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
            int indx1);
    void svPutLogicArrElem2VecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
            int indx1, int indx2);
    void svPutLogicArrElem3VecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
            int indx1, int indx2, int indx3);

    /* From simulator storage into user space */
    void svGetBitArrElemVecVal(svBitVecVal* d, const svOpenArrayHandle s,
            int indx1, ...);
    void svGetBitArrElem1VecVal(svBitVecVal* d, const svOpenArrayHandle s,
            int indx1);
    void svGetBitArrElem2VecVal(svBitVecVal* d, const svOpenArrayHandle s,
            int indx1, int indx2);
    void svGetBitArrElem3VecVal(svBitVecVal* d, const svOpenArrayHandle s,
            int indx1, int indx2, int indx3);
    void svGetLogicArrElemVecVal(svLogicVecVal* d, const svOpenArrayHandle s,
            int indx1, ...);
    void svGetLogicArrElem1VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
            int indx1);
    void svGetLogicArrElem2VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
            int indx1, int indx2);
    void svGetLogicArrElem3VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
            int indx1, int indx2, int indx3);
```

The above functions copy the whole packed array in either direction. The user is responsible for allocating an array in the canonical representation.

F.11.6 Access to scalar elements (bit and logic)

Another group of functions is needed for scalars (i.e., when an element of an array is a simple scalar, **bit**, or **logic**):

```
svBit    svGetBitArrElem (const svOpenArrayHandle s, int indx1, ...);
svBit    svGetBitArrElem1(const svOpenArrayHandle s, int indx1);
svBit    svGetBitArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svBit    svGetBitArrElem3(const svOpenArrayHandle s, int indx1, int indx2,
                              int indx3);

svLogic svGetLogicArrElem (const svOpenArrayHandle s, int indx1, ...);
svLogic svGetLogicArrElem1(const svOpenArrayHandle s, int indx1);
svLogic svGetLogicArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svLogic svGetLogicArrElem3(const svOpenArrayHandle s, int indx1, int indx2,
                              int indx3);

void svPutLogicArrElem (const svOpenArrayHandle d, svLogic value, int indx1,
                              ...);
void svPutLogicArrElem1(const svOpenArrayHandle d, svLogic value, int indx1);
void svPutLogicArrElem2(const svOpenArrayHandle d, svLogic value, int indx1,
                              int indx2);
void svPutLogicArrElem3(const svOpenArrayHandle d, svLogic value, int indx1,
                              int indx2, int indx3);

void svPutBitArrElem (const svOpenArrayHandle d, svBit value, int indx1,
...);
void svPutBitArrElem1(const svOpenArrayHandle d, svBit value, int indx1);
void svPutBitArrElem2(const svOpenArrayHandle d, svBit value, int indx1,
                              int indx2);
void svPutBitArrElem3(const svOpenArrayHandle d, svBit value, int indx1,
                              int indx2, int indx3);
```

F.11.7 Access to array elements of other types

If an array's elements are of a type compatible with C, there is no need to use canonical representation. In such situations, the elements are accessed via pointers, i.e., the actual address of an element shall be computed first and then used to access the desired element.

F.11.8 Example 6—Two-dimensional open array

SystemVerilog:

```
typedef struct {int i; ... } MyType;

import "DPI-C" function void foo(input MyType i [][]); /* 2-dimensional unsized
                                                unpacked array of MyType */

MyType a_10x5 [11:20][6:2];
MyType a_64x8 [64:1][-1:-8];

foo(a_10x5);
foo(a_64x8);
```

C:

```
#include "svdpi.h"

typedef struct {int i; ... } MyType;

void foo(const svOpenArrayHandle h)
{
   MyType my_value;
   int i, j;
   int lo1 = svLow(h, 1);
   int hi1 = svHigh(h, 1);
   int lo2 = svLow(h, 2);
   int hi2 = svHigh(h, 2);

   for (i = lo1; i <= hi1; i++) {
      for (j = lo2; j <= hi2; j++) {

         my_value = *(MyType *)svGetArrElemPtr2(h, i, j);
         ...
         *(MyType *)svGetArrElemPtr2(h, i, j) = my_value;
         ...
      }
   ...
   }
}
```

F.11.9 Example 7—Open array

SystemVerilog:

```
typedef struct { ... } MyType;

import "DPI-C" function void foo(input MyType i [], output MyType o []);

MyType source [11:20];
MyType target [11:20];

foo(source, target);
```

C:

```
#include "svdpi.h"

typedef struct ... } MyType;

void foo(const svOpenArrayHandle hin, const svOpenArrayHandle hout)
{
   int count = svSize(hin, 1);
   MyType *s = (MyType *)svGetArrayPtr(hin);
   MyType *d = (MyType *)svGetArrayPtr(hout);

   if (s && d) { /* both arrays have C layout */

   /* an efficient solution using pointer arithmetic */
   while (count--)
      *d++ = *s++;

   /* even more efficient:
```

```
            memcpy(d, s, svSizeOfArray(hin));
      */

   } else { /* less efficient yet implementation independent */

            int i = svLow(hin, 1);
            int j = svLow(hout, 1);
            while (i <= svHigh(hin, 1)) {
               *(MyType *)svGetArrElemPtr1(hout, j++) =
               *(MyType *)svGetArrElemPtr1(hin, i++);
            }

   }

}
```

F.11.10 Example 8—Access to packed arrays

SystemVerilog:

```
   import "DPI-C" function void foo(input logic [127:0]);
   import "DPI-C" function void boo(input logic [127:0] i []); // open array of
                                                               // 128-bit
```

C:

```
   #include "svdpi.h"

   /* Copy out one 128-bit packed vector */
   void foo(const svLogicVecVal* packed_vec_128_bit)
   {
      svLogicVecVal arr[SV_PACKED_DATA_NELEMS(128)]; /* canonical rep */
      memcpy(arr, packed_vec_128_bit, sizeof(arr));
      ...
   }

   * Copy out each word of an open array of 128-bit packed vectors */
   void boo(const svOpenArrayHandle h)
   {
      int i;
      svLogicVecVal arr[SV_PACKED_DATA_NELEMS(128)]; /* canonical rep */
      for (i = svLow(h, 1); i <= svHigh(h, 1); i++) {
         const svLogicVecVal* ptr = (svLogicVecVal*)svGetArrElemPtr1(h, i);
         memcpy(arr, ptr, sizeof(arr));
         ...
      }
      ...
   }
```

F.12 SV3.1a-compatible access to packed data (deprecated functionality)

The functionality described in this subclause is deprecated and need not be implemented by an IEEE 1800
simulator. The functionality provides backwards compatibility with Accellera SystemVerilog 3.1a (SV3.1a)
[B1] regarding the semantics of packed array arguments. This subclause will describe the SV3.1a semantics.

The main difference between SV3.1a and IEEE 1800 semantics is that in SV3.1a, packed data arguments are passed by opaque handle types `svLogicPackedArrRef` and `svBitPackedArrRef`. An implementation need not do any conversion or marshalling of data into the canonical format. The C programmer is provided a set of utility functions that copies data between actual vendor format and canonical format. Other utilities are provided that put and get bit-selects and part-selects from actual vendor representation.

F.12.1 Determining the compatibility level of an implementation

Function `svDpiVersion()` is provided to allow the determination of an implementation's support for this standard. In simulators that only support the SV3.1a standard, users must make use of the opaque handle types for all 2-state and 4-state arguments. See F.9.1.3.

When using an IEEE 1800 implementation, it is possible for users to make use of SV3.1a-compatible semantics on a per-function basis. Import and export declarations annotated with the `"DPI"` syntax shall yield the SV3.1a argument passing semantics on the C side of the interface. Import and export declarations annotated with the `"DPI-C"` syntax shall yield the IEEE 1800 argument passing semantics. See 26.3 and 26.4.4.

The `svdpi.h` file may contain definitions and function prototypes for use with SV3.1a-compliant packed data access. IEEE 1800 implementations are not obligated to provide these definitions and prototypes in the include file.

If an IEEE 1800 implementation does not support the functionality in this subclause, it is possible that the DPI C code may not successfully bind to the implementation.

F.12.2 svdpi.h definitions for SV3.1a-style packed data processing

The following definitions are used to define SV3.1a-style canonical access to packed data.

```
/* 2-state and 4-state vectors, modeled upon PLI's avalue/bvalue */
#define SV_CANONICAL_SIZE(WIDTH) (((WIDTH)+31)>>5)

typedef uint32_t
   svBitVec32; /* (a chunk of) packed bit array */

typedef struct { unsigned int c; unsigned int d;}
   svLogicVec32; /* (a chunk of) packed logic array */
```

The following definitions describe implementation-dependent packed data representation.

```
/* reference to a standalone packed array */
typedef void* svBitPackedArrRef;
typedef void* svLogicPackedArrRef;

/* total size in bytes of the simulator's representation of a packed array */
/* width in bits */
int svSizeOfBitPackedArr(int width);

int svSizeOfLogicPackedArr(int width);
```

The following functions provide translation between actual vendor representation and canonical representation. The functions copy the whole array in either direction. The user is responsible for providing the correct width and for allocating an array in the canonical representation. The contents of the unused bits are undetermined.

Although the put and get functionality provided for **bit** and **logic** packed arrays is sufficient, yet basic, it requires unnecessary copying of the whole packed array when perhaps only some bits are needed. For the sake of convenience and improved performance, bit-selects and limited (up to 32 bits) part-selects are also supported.

```
/* s=source, d=destination, w=width */
/* actual <-- canonical */
void svPutBitVec32 (svBitPackedArrRef d, const svBitVec32* s, int w);
void svPutLogicVec32 (svLogicPackedArrRef d, const svLogicVec32* s, int w);

/* canonical <-- actual */
void svGetBitVec32 (svBitVec32* d, const svBitPackedArrRef s, int w);
void svGetLogicVec32 (svLogicVec32* d, const svLogicPackedArrRef s, int w);
```

The following functions provide support for bit-select processing on actual vendor data representation.

```
/* Packed arrays are assumed to be indexed n-1:0, where 0 is the index of
   LSB */
/* functions for bit-select */
/* s=source, i=bit-index */
svBit svGetSelectBit(const svBitPackedArrRef s, int i);
svLogic svGetSelectLogic(const svLogicPackedArrRef s, int i);

/* d=destination, i=bit-index, s=scalar */
void svPutSelectBit(svBitPackedArrRef d, int i, svBit s);
void svPutSelectLogic(svLogicPackedArrRef d, int i, svLogic s);
```

Limited (up to 32-bit) part-selects are supported. A part-select is a slice of a packed array of types **bit** or **logic**. Array slices are not supported for unpacked arrays. Functions for part-selects only allow access (read/write) to a narrow subrange of up to 32 bits. Canonical representation shall be used for such narrow vectors. If the specified range of a part-select is not fully contained within the normalized range of an array, the behavior is undetermined.

```
/*
 * functions for part-select
 *
 * a narrow (<=32 bits) part-select is copied between
 * the implementation representation and a single chunk of
 * canonical representation
 * Normalized ranges and indexing [n-1:0] are used for both arrays:
 * the array in the implementation representation and the canonical array.
 *
 * s=source, d=destination, i=starting bit index, w=width
 * like for variable part-selects; limitations: w <= 32
 *
 * In part-select operations, the data are copied to or from the
 * canonical representation part ('chunk') designated by range [w-1:0]
 * and the implementation representation part designated by range [w+i-1:i].
 */

/* canonical <-- actual */
void svGetPartSelectBit(svBitVec32* d, const svBitPackedArrRef s, int i,
        int w);
svBitVec32 svGetBits(const svBitPackedArrRef s, int i, int w);
svBitVec32 svGet32Bits(const svBitPackedArrRef s, int i); // 32-bits
uint64_t svGet64Bits(const svBitPackedArrRef s, int i); // 64-bits
void svGetPartSelectLogic(svLogicVec32* d, const svLogicPackedArrRef s, int
i,
```

```
        int w);

    /* actual <-- canonical */
    void svPutPartSelectBit(svBitPackedArrRef d, const svBitVec32 s, int i,
        int w);
    void svPutPartSelectLogic(svLogicPackedArrRef d, const svLogicVec32 s, int i,
        int w);
```

F.12.3 Source-level compatibility include file svdpi_src.h

Only two symbols are defined: the macros that allow declaring variables to represent the SystemVerilog packed arrays of type **bit** or **logic**. Applications that do not need this file to compile are deemed binary-compatible. In other words, the DPI C code does not need to be recompiled to run on different simulators. Applications that make use of svdpi_src.h must be recompiled for each simulator on which they are to be run.

```
    #define SV_BIT_PACKED_ARRAY(WIDTH,NAME) ...
    #define SV_LOGIC_PACKED_ARRAY(WIDTH,NAME) ...
```

The actual definitions are implementation-specific, but must not define an array type (see definition in 6.2.5 in ISO 9899-2001 [B5]). For example, a SystemVerilog simulator might define the latter macro as follows:

```
    #define SV_LOGIC_PACKED_ARRAY(WIDTH,NAME) \
        struct { svLogicVec32 __unnamed [SV_CANONICAL_SIZE(WIDTH)]; } NAME
```

F.12.4 Example 9—Deprecated SV3.1a binary compatible application

SystemVerilog:

```
    typedef struct {int x; int y;} pair;
    import "DPI" function void foo(input int i1, pair i2, output logic [63:0]
    o3);

    export "DPI" function exported_sv_func;

    function void exported_sv_func(input int i, output int o [0:7]);
        begin ... end
    endfunction
```

C:

```
    include "svdpi.h"
    typedef struct {int x; int y;} pair;
    extern void exported_sv_func(int, int *); /* imported from SystemVerilog */
    void foo(const int i1, const pair *i2, svLogicPackedArrRef* o3)
    {
        svLogicVec32 arr[SV_CANONICAL_SIZE(64)]; /* 2 chunks needed */
        int tab[8];
        printf("%d\n", i1);
        arr[0].c = i2->x;
        arr[0].d = 0;
        arr[1].c = i2->y;
        arr[1].d = 0;
        svPutLogicVec32(o3, arr, 64);
```

```
        /* call SystemVerilog */
        exported_sv_func(i1, tab); /* tab passed by reference */
        ...
    }
```

F.12.5 Example 10—Deprecated SV3.1a source compatible application

SystemVerilog:

```
    typedef struct {int a; bit [6:1][1:8] b [65:2]; int c;} triple;
    // troublesome mix of C types and packed arrays
    import "DPI" function void foo(input triple t);

    export "DPI" function exported_sv_func;

    function void exported_sv_func(input int i, output logic [63:0] o);
        begin ... end
    endfunction
```

C:

```
    include "svdpi.h"
    #include "svdpi_src.h"

    typedef struct {
        int a;
        SV_BIT_PACKED_ARRAY(6*8, b) [64]; /* implementation-specific
    representation                                              */
        int c;
    } triple;
    /* Note that 'b' is defined as for 'bit [6*8-1:0] b [63:0]' */

    extern void exported_sv_func(int, svLogicPackedArrRef); /* imported from
                                                    SystemVerilog */

    void foo(const triple *t)
    {
        int j;
        /* canonical representation */
        svBitVec32   aB[SV_CANONICAL_SIZE(6*8)]; /* 6*8 packed bits */
        svLogicVec32 aL[SV_CANONICAL_SIZE(64)];

        /* implementation-specific representation */
        SV_LOGIC_PACKED_ARRAY(64, my_tab);

        printf("%d %d\n", t->a, t->c);
        for (i = 0; i < 64; i++) {
            svGetBitVec32(aB, (svBitPackedArrRef)&(t->b[i]), 6*8);
            ...
        }
        ...
        /* call SystemVerilog */
        exported_sv_func(2, (svLogicPackedArrRef)&my_tab); /* by reference */
        svGetLogicVec32(aL, (svLogicPackedArrRef)&my_tab, 64);
        ...
    }
```

F.12.6 Example 11—Deprecated SV3.1a binary compatible calls of export functions

This example demonstrates that the source compatibility include file `svdpi_src.h` is not needed if a C function dynamically allocates the data structure for simulator representation of a packed array to be passed to an exported SystemVerilog function.

SystemVerilog:

```
export "DPI" function myfunc;
...
function void myfunc (output logic [31:0] r); ...
...
```

C:

```
include "svdpi.h"
extern void myfunc (svLogicPackedArrRef r); /* exported from SV */

/* output logic packed 32-bits */
...
svLogicVec32 my_r[SV_CANONICAL_SIZE(32)];
/* my array, canonical representation */

/* allocate memory for logic packed 32-bits in simulator's representation */
svLogicPackedArrRef r =
    (svLogicPackedArrRef)malloc(svSizeOfLogicPackedArr(32));
myfunc(r);
/* canonical <-- actual */
svGetLogicVec32(my_r, r, 32);
/* shall use only the canonical representation from now on */
free(r); /* do not need any more */
...
```

Annex G

(normative)

Include file svdpi.h

This annex shows the contents of the `svdpi.h` include file. This is a normative include file that must be provided by all SystemVerilog simulators. However, there is deprecated functionality at the bottom of the file that need not be provided. This functionality is clearly delimited by comments in the file.

Implementations shall ensure that types `uint8_t` and `uint32_t` are defined, but the exact method of doing so is not prescribed by this standard. The section in the include file shown below is a suggested way of defining `uint8_t` and `uint32_t` for a wide variety of SystemVerilog platforms.

```
/*
 * svdpi.h
 *
 * SystemVerilog Direct Programming Interface (DPI).
 *
 * This file contains the constant definitions, structure definitions,
 * and routine declarations used by SystemVerilog DPI.
 */

#ifndef INCLUDED_SVDPI
#define INCLUDED_SVDPI

#ifdef __cplusplus
extern "C" {
#endif

/* Ensure that size-critical types are defined on all OS platforms. */
#if defined (_MSC_VER)
typedef unsigned __int64 uint64_t;
typedef unsigned __int32 uint32_t;
typedef unsigned __int8 uint8_t;
typedef signed __int64 int64_t;
typedef signed __int32 int32_t;
typedef signed __int8 int8_t;
#elif defined(__MINGW32__)
#include <stdint.h>
#elif defined(__linux)
#include <inttypes.h>
#else
#include <sys/types.h>
#endif

/* Use to export a symbol from application */
#if defined (_MSC_VER)
#define DPI_DLLISPEC __declspec(dllimport)
#else
#define DPI_DLLISPEC
#endif

/* Use to import a symbol into application */
#if defined (_MSC_VER)
#define DPI_DLLESPEC __declspec(dllexport)
#else
#define DPI_DLLESPEC
```

```
#endif

/* Use to mark a function as external */
#ifndef DPI_EXTERN
#define DPI_EXTERN
#endif

#ifndef DPI_PROTOTYPES
#define DPI_PROTOTYPES
/* object is defined imported by the application */
#define XXTERN DPI_EXTERN DPI_DLLISPEC
/* object is exported by the application */
#define EETERN DPI_EXTERN DPI_DLLESPEC
#endif


/* canonical representation */
#define sv_0 0
#define sv_1 1
#define sv_z 2
#define sv_x 3

/* common type for 'bit' and 'logic' scalars. */
typedef uint8_t svScalar;
typedef svScalar svBit; /* scalar */
typedef svScalar svLogic; /* scalar */



/*
 * DPI representation of packed arrays.
 * 2-state and 4-state vectors, exactly the same as PLI's avalue/bvalue.
 */
#ifndef VPI_VECVAL
#define VPI_VECVAL
typedef struct vpi_vecval {
    uint32_t a;
    uint32_t b;
} s_vpi_vecval, *p_vpi_vecval;
#endif

/* (a chunk of) packed logic array */
typedef s_vpi_vecval svLogicVecVal;

/* (a chunk of) packed bit array */
typedef uint32_t svBitVecVal;

/* Number of chunks required to represent the given width packed array */
#define SV_PACKED_DATA_NELEMS(WIDTH) (((WIDTH) + 31) >> 5)

/*
 * Because the contents of the unused bits is undetermined,
 * the following macros can be handy.
 */
#define SV_MASK(N) (~(-1 << (N)))

#define SV_GET_UNSIGNED_BITS(VALUE, N) \
    ((N) == 32 ? (VALUE) : ((VALUE) & SV_MASK(N)))

#define SV_GET_SIGNED_BITS(VALUE, N) \
    ((N) == 32 ? (VALUE) : \
```

```
    (((VALUE) & (1 << (N))) ? ((VALUE) | ~SV_MASK(N)) : ((VALUE) & SV_MASK(N))))


/*
 * Implementation-dependent representation.
 */
/*
 * Return implementation version information string ("P1800-2005" or "SV3.1a").
 */
XXTERN const char* svDpiVersion();

/* a handle to a scope (an instance of a module or interface) */
XXTERN typedef void* svScope;

/* a handle to a generic object (actually, unsized array) */
XXTERN typedef void* svOpenArrayHandle;

/*
 * Bit-select utility functions.
 *
 * Packed arrays are assumed to be indexed n-1:0,
 * where 0 is the index of LSB
 */

/* s=source, i=bit-index */
XXTERN svBit svGetBitselBit(const svBitVecVal* s, int i);
XXTERN svLogic svGetBitselLogic(const svLogicVecVal* s, int i);

/* d=destination, i=bit-index, s=scalar */
XXTERN void svPutBitselBit(svBitVecVal* d, int i, svBit s);
XXTERN void svPutBitselLogic(svLogicVecVal* d, int i, svLogic s);

/*
 * Part-select utility functions.
 *
 * A narrow (<=32 bits) part-select is extracted from the
 * source representation and written into the destination word.
 *
 * Normalized ranges and indexing [n-1:0] are used for both arrays.
 *
 * s=source, d=destination, i=starting bit index, w=width
 * like for variable part-selects; limitations: w <= 32
 */
XXTERN void svGetPartselBit(svBitVecVal* d, const svBitVecVal* s, int i, int w);
XXTERN void svGetPartselLogic(svLogicVecVal* d, const svLogicVecVal* s, int i, int
w);

XXTERN void svPutPartselBit(svBitVecVal* d, const svBitVecVal s, int i, int w);
XXTERN void svPutPartselLogic(svLogicVecVal* d, const svLogicVecVal s, int i, int w);


/*
 * Open array querying functions
 * These functions are modeled upon the SystemVerilog array
 * querying functions and use the same semantics.
 *
 * If the dimension is 0, then the query refers to the
 * packed part of an array (which is one-dimensional).
 * Dimensions > 0 refer to the unpacked part of an array.
 */
```

```
/* h= handle to open array, d=dimension */
XXTERN int svLeft(const svOpenArrayHandle h, int d);
XXTERN int svRight(const svOpenArrayHandle h, int d);
XXTERN int svLow(const svOpenArrayHandle h, int d);
XXTERN int svHigh(const svOpenArrayHandle h, int d);
XXTERN int svIncrement(const svOpenArrayHandle h, int d);
XXTERN int svLength(const svOpenArrayHandle h, int d);
XXTERN int svDimensions(const svOpenArrayHandle h);

/*
 * Pointer to the actual representation of the whole array of any type
 * NULL if not in C layout
 */
XXTERN void *svGetArrayPtr(const svOpenArrayHandle);

/* total size in bytes or 0 if not in C layout */
XXTERN int svSizeOfArray(const svOpenArrayHandle);

/*
 * Return a pointer to an element of the array
 * or NULL if index outside the range or null pointer
 */
XXTERN void *svGetArrElemPtr(const svOpenArrayHandle, int indx1, ...);

/* specialized versions for 1-, 2- and 3-dimensional arrays: */
XXTERN void *svGetArrElemPtr1(const svOpenArrayHandle, int indx1);
XXTERN void *svGetArrElemPtr2(const svOpenArrayHandle, int indx1, int indx2);
XXTERN void *svGetArrElemPtr3(const svOpenArrayHandle, int indx1, int indx2,
        int indx3);

/*
 * Functions for copying between simulator storage and user space.
 * These functions copy the whole packed array in either direction.
 * The user is responsible for allocating an array to hold the
 * canonical representation.
 */

/* s=source, d=destination */
/* From user space into simulator storage */
XXTERN void svPutBitArrElemVecVal(const svOpenArrayHandle d, const svBitVecVal* s,
        int indx1, ...);
XXTERN void svPutBitArrElem1VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
        int indx1);
XXTERN void svPutBitArrElem2VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
        int indx1, int indx2);
XXTERN void svPutBitArrElem3VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
        int indx1, int indx2, int indx3);
XXTERN void svPutLogicArrElemVecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
        int indx1, ...);
XXTERN void svPutLogicArrElem1VecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
        int indx1);
XXTERN void svPutLogicArrElem2VecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
        int indx1, int indx2);
XXTERN void svPutLogicArrElem3VecVal(const svOpenArrayHandle d, const svLogicVecVal* s,
        int indx1, int indx2, int indx3);


/* From simulator storage into user space */
XXTERN void svGetBitArrElemVecVal(svBitVecVal* d, const svOpenArrayHandle s,
        int indx1, ...);
XXTERN void svGetBitArrElem1VecVal(svBitVecVal* d, const svOpenArrayHandle s,
```

```
        int indx1);
XXTERN void svGetBitArrElem2VecVal(svBitVecVal* d, const svOpenArrayHandle s,
        int indx1, int indx2);
XXTERN void svGetBitArrElem3VecVal(svBitVecVal* d, const svOpenArrayHandle s,
        int indx1, int indx2, int indx3);
XXTERN void svGetLogicArrElemVecVal(svLogicVecVal* d, const svOpenArrayHandle s,
        int indx1, ...);
XXTERN void svGetLogicArrElem1VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
        int indx1);
XXTERN void svGetLogicArrElem2VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
        int indx1, int indx2);
XXTERN void svGetLogicArrElem3VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
        int indx1, int indx2, int indx3);


XXTERN svBit svGetBitArrElem(const svOpenArrayHandle s, int indx1, ...);
XXTERN svBit svGetBitArrElem1(const svOpenArrayHandle s, int indx1);
XXTERN svBit svGetBitArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
XXTERN svBit svGetBitArrElem3(const svOpenArrayHandle s, int indx1, int indx2,
        int indx3);
XXTERN svLogic svGetLogicArrElem(const svOpenArrayHandle s, int indx1, ...);
XXTERN svLogic svGetLogicArrElem1(const svOpenArrayHandle s, int indx1);
XXTERN svLogic svGetLogicArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
XXTERN svLogic svGetLogicArrElem3(const svOpenArrayHandle s, int indx1, int indx2,
        int indx3);
XXTERN void svPutLogicArrElem(const svOpenArrayHandle d, svLogic value, int indx1,
        ...);
XXTERN void svPutLogicArrElem1(const svOpenArrayHandle d, svLogic value, int indx1);
XXTERN void svPutLogicArrElem2(const svOpenArrayHandle d, svLogic value, int indx1,
        int indx2);
XXTERN void svPutLogicArrElem3(const svOpenArrayHandle d, svLogic value, int indx1,
        int indx2, int indx3);
XXTERN void svPutBitArrElem(const svOpenArrayHandle d, svBit value, int indx1, ...);
XXTERN void svPutBitArrElem1(const svOpenArrayHandle d, svBit value, int indx1);
XXTERN void svPutBitArrElem2(const svOpenArrayHandle d, svBit value, int indx1,
        int indx2);
XXTERN void svPutBitArrElem3(const svOpenArrayHandle d, svBit value, int indx1,
        int indx2, int indx3);


/* Functions for working with DPI context */

/*
 * Retrieve the active instance scope currently associated with the executing
 * imported function. Unless a prior call to svSetScope has occurred, this
 * is the scope of the function's declaration site, not call site.
 * Returns NULL if called from C code that is *not* an imported function.
 */
XXTERN svScope svGetScope();

/*
 * Set context for subsequent export function execution.
 * This function must be called before calling an export function, unless
 * the export function is called while executing an extern function. In that
 * case the export function shall inherit the scope of the surrounding extern
 * function. This is known as the "default scope".
 * The return is the previous active scope (per svGetScope)
 */
XXTERN svScope svSetScope(const svScope scope);

/* Gets the fully qualified name of a scope handle */
XXTERN const char* svGetNameFromScope(const svScope);
```

```
/*
 * Retrieve svScope to instance scope of an arbitrary function declaration.
 * (can be either module, program, interface, or generate scope)
 * The return value shall be NULL for unrecognized scope names.
 */
XXTERN svScope svGetScopeFromName(const char* scopeName);

/*
 * Store an arbitrary user data pointer for later retrieval by svGetUserData()
 * The userKey is generated by the user. It must be guaranteed by the user to
 * be unique from all other userKey's for all unique data storage requirements
 * It is recommended that the address of static functions or variables in the
 * user's C code be used as the userKey.
 * It is illegal to pass in NULL values for either the scope or userData
 * arguments. It is also an error to call svPutUserData() with an invalid
 * svScope. This function returns -1 for all error cases, 0 upon success. It is
 * suggested that userData values of 0 (NULL) not be used as otherwise it can
 * be impossible to discern error status returns when calling svGetUserData()
 */
XXTERN int svPutUserData(const svScope scope, void *userKey, void* userData);

/*
 * Retrieve an arbitrary user data pointer that was previously
 * stored by a call to svPutUserData(). See the comment above
 * svPutUserData() for an explanation of userKey, as well as
 * restrictions on NULL and illegal svScope and userKey values.
 * This function returns NULL for all error cases, 0 upon success.
 * This function also returns NULL in the event that a prior call
 * to svPutUserData() was never made.
 */
XXTERN void* svGetUserData(const svScope scope, void* userKey);

/*
 * Returns the file and line number in the SV code from which the extern call
 * was made. If this information available, returns TRUE and updates fileName
 * and lineNumber to the appropriate values. Behavior is unpredictable if
 * fileName or lineNumber are not appropriate pointers. If this information is
 * not available return FALSE and contents of fileName and lineNumber not
 * modified. Whether this information is available or not is implementation-
 * specific. Note that the string provided (if any) is owned by the SV
 * implementation and is valid only until the next call to any SV function.
 * Applications must not modify this string or free it
 */
XXTERN int svGetCallerInfo(const char** fileName, int *lineNumber);

/*
 * Returns 1 if the current execution thread is in the disabled state.
 * Disable protocol must be adhered to if in the disabled state.
 */
XXTERN int svIsDisabledState();

/*
 * Imported functions call this API function during disable processing to
 * acknowledge that they are correctly participating in the DPI disable protocol.
 * This function must be called before returning from an imported function that is
 * in the disabled state.
 */
XXTERN void svAckDisabledState();
```

```
/*
 *************************************************************
 * DEPRECATED PORTION OF FILE STARTS FROM HERE.
 * IEEE-P1800-compliant tools may not provide
 * support for the following functionality.
 *************************************************************
 */

/*
 * Canonical representation of packed arrays
 * 2-state and 4-state vectors, modeled upon PLI's avalue/bvalue
 */
#define SV_CANONICAL_SIZE(WIDTH) (((WIDTH)+31)>>5)
typedef unsigned int svBitVec32;/* (a chunk of) packed bit array */
typedef struct { unsigned int c; unsigned int d;}
svLogicVec32; /* (a chunk of) packed logic array */

/* reference to a standalone packed array */
typedef void* svBitPackedArrRef;
typedef void* svLogicPackedArrRef;

/*
 * total size in bytes of the simulator's representation of a packed array
 * width in bits
 */
XXTERN int svSizeOfBitPackedArr(int width);
XXTERN int svSizeOfLogicPackedArr(int width);

/* Translation between the actual representation and the canonical representation */

/* s=source, d=destination, w=width */
/* actual <-- canonical */
XXTERN void svPutBitVec32(svBitPackedArrRef d, const svBitVec32* s, int w);
XXTERN void svPutLogicVec32(svLogicPackedArrRef d, const svLogicVec32* s, int w);

/* canonical <-- actual */
XXTERN void svGetBitVec32(svBitVec32* d, const svBitPackedArrRef s, int w);
XXTERN void svGetLogicVec32(svLogicVec32* d, const svLogicPackedArrRef s, int w);

/*
 * Bit-select functions
 * Packed arrays are assumed to be indexed n-1:0,
 * where 0 is the index of LSB
 */

/* s=source, i=bit-index */
XXTERN svBit svGetSelectBit(const svBitPackedArrRef s, int i);
XXTERN svLogic svGetSelectLogic(const svLogicPackedArrRef s, int i);

/* d=destination, i=bit-index, s=scalar */
XXTERN void svPutSelectBit(svBitPackedArrRef d, int i, svBit s);
XXTERN void svPutSelectLogic(svLogicPackedArrRef d, int i, svLogic s);

/*
 * functions for part-select
 *
 * a narrow (<=32 bits) part-select is copied between
 * the implementation representation and a single chunk of
 * canonical representation
 * Normalized ranges and indexing [n-1:0] are used for both arrays:
```

613

```
 * the array in the implementation representation and the canonical array.
 *
 * s=source, d=destination, i=starting bit index, w=width
 * like for variable part-selects; limitations: w <= 32
 */

/* canonical <-- actual */
XXTERN void svGetPartSelectBit(svBitVec32* d, const svBitPackedArrRef s,
        int i, int w);
XXTERN svBitVec32 svGetBits(const svBitPackedArrRef s, int i, int w);
XXTERN svBitVec32 svGet32Bits(const svBitPackedArrRef s, int i); /* 32-bits */

XXTERN uint64_t svGet64Bits(const svBitPackedArrRef s, int i);

/* 64-bits */
XXTERN void svGetPartSelectLogic(svLogicVec32* d, const svLogicPackedArrRef s,
        int i, int w);
/* actual <-- canonical */
XXTERN void svPutPartSelectBit(svBitPackedArrRef d, const svBitVec32 s,
        int i, int w);
XXTERN void svPutPartSelectLogic(svLogicPackedArrRef d, const svLogicVec32 s,
        int i, int w);

/*
 * Functions for open array translation between simulator and canonical
representations.
 * These functions copy the whole packed array in either direction. The user is
 * responsible for allocating an array in the canonical representation.
 */

/* s=source, d=destination */
/* actual <-- canonical */
XXTERN void svPutBitArrElemVec32(const svOpenArrayHandle d, const svBitVec32* s,
        int indx1, ...);
XXTERN void svPutBitArrElem1Vec32(const svOpenArrayHandle d, const svBitVec32* s,
        int indx1);
XXTERN void svPutBitArrElem2Vec32(const svOpenArrayHandle d, const svBitVec32* s,
        int indx1, int indx2);
XXTERN void svPutBitArrElem3Vec32(const svOpenArrayHandle d, const svBitVec32* s,
        int indx1, int indx2, int indx3);
XXTERN void svPutLogicArrElemVec32(const svOpenArrayHandle d, const svLogicVec32* s,
        int indx1, ...);
XXTERN void svPutLogicArrElem1Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
        int indx1);
XXTERN void svPutLogicArrElem2Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
        int indx1, int indx2);
XXTERN void svPutLogicArrElem3Vec32(const svOpenArrayHandle d, const svLogicVec32* s,
        int indx1, int indx2, int indx3);

/* canonical <-- actual */
XXTERN void svGetBitArrElemVec32(svBitVec32* d, const svOpenArrayHandle s,
        int indx1, ...);
XXTERN void svGetBitArrElem1Vec32(svBitVec32* d, const svOpenArrayHandle s,
        int indx1);
XXTERN void svGetBitArrElem2Vec32(svBitVec32* d, const svOpenArrayHandle s,
        int indx1, int indx2);
XXTERN void svGetBitArrElem3Vec32(svBitVec32* d, const svOpenArrayHandle s,
        int indx1, int indx2, int indx3);
XXTERN void svGetLogicArrElemVec32(svLogicVec32* d, const svOpenArrayHandle s,
        int indx1, ...);
XXTERN void svGetLogicArrElem1Vec32(svLogicVec32* d, const svOpenArrayHandle s,
```

```
        int indx1);
XXTERN void svGetLogicArrElem2Vec32(svLogicVec32* d, const svOpenArrayHandle s,
        int indx1, int indx2);
XXTERN void svGetLogicArrElem3Vec32(svLogicVec32* d, const svOpenArrayHandle s,
        int indx1, int indx2, int indx3);

/*
 **********************************************************
 * DEPRECATED PORTION OF FILE ENDS HERE.
 **********************************************************
 */

#undef DPI_EXTERN

#ifdef DPI_PROTOTYPES
#undef DPI_PROTOTYPES
#undef XXTERN
#undef EETERN
#endif


#ifdef __cplusplus
}
#endif

#endif
```

Annex H

(normative)

Inclusion of foreign language code

H.1 General

This annex describes common guidelines for the inclusion of foreign language code into a SystemVerilog application. The intention of these guidelines is to enable the redistribution of C binaries in shared object form.

Foreign language code is functionality that is included into SystemVerilog using the DPI. As a result, all statements of this annex apply only to code included using this interface; code included by using other interfaces (e.g., PLI or VPI) is outside the scope of this standard. Due to the nature of the DPI, most foreign language code is usually created from C or C++ source code, although nothing precludes the creation of appropriate object code from other languages. This annex adheres to this rule: its content is independent from the actual language used.

In general, foreign language code is provided in the form of object code compiled for the actual platform. The capability to include foreign language code in object-code form shall be supported by all simulators as specified here.

This annex defines how to

— Specify the location of the corresponding files within the file system.
— Specify the files to be loaded (in case of object code).
— Provide the object code (as a shared library or archive).

Although this annex defines guidelines for a common inclusion methodology, it requires multiple implementations (usually two) of the corresponding facilities. This takes into account that multiple users can have different viewpoints and different requirements on the inclusion of foreign language code.

— A vendor that wants to provide its Internet Protocol (IP) in the form of foreign language code often requires a self-contained method for the integration, which still permits an integration by a third party. This use case is often covered by a bootstrap file approach.
— A project team that specifies a common, standard set of foreign language code might change the code depending on technology, selected cells, back-annotation data, and other items. This use case is often covered by a set of tool switches, although it might also use the bootstrap file approach.
— An user might want to switch between selections or provide additional code. This use case is covered by providing a set of tool switches to define the corresponding information, although it might also use the bootstrap file approach.

NOTE—This annex defines a set of switch names to be used for a particular functionality. This is of informative nature; the actual naming of switches is not part of this standard. Further, it might not be possible to use certain character configurations in all operating systems or shells. Therefore, any switch name defined within this standard is a recommendation on how to name a switch, but not a requirement of the language.

H.2 Location independence

All pathnames specified within this annex are intended to be location independent, which is accomplished by using the switch `-sv_root`. It can receive a single directory pathname as the value, which is then

prepended to any relative pathname that has been specified. In absence of this switch, or when processing relative filenames before any `-sv_root` specification, the current working directory of the user shall be used as the default value.

H.3 Object code inclusion

Compiled object code is required for cases where the compilation and linking of source code are fully handled by the user; thus, the created object code only need be loaded to integrate the foreign language code into a SystemVerilog application. All SystemVerilog applications shall support the integration of foreign language code in object code form. Figure H-1 depicts the inclusion of object code and its relations to the various steps involved in this integration process.
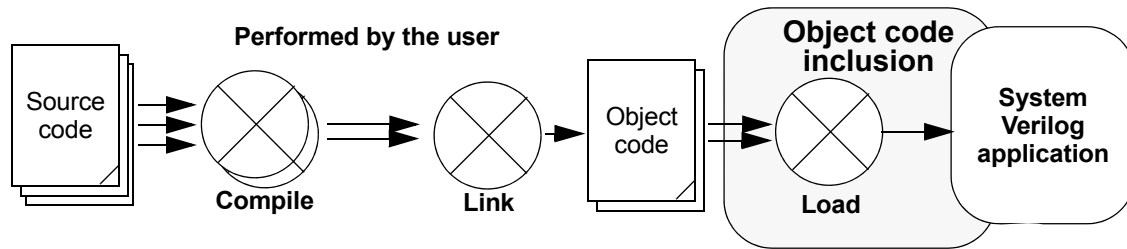
Figure H-1—Inclusion of object code into a SystemVerilog application

Compiled object code can be specified by one of the following two methods:

a) By an entry in a bootstrap file; see H.3.1 for more details on this file and its content. Its location shall be specified with one instance of the switch `-sv_liblist` *pathname*. This switch can be used multiple times to define the usage of multiple bootstrap files.

b) By specifying the file with one instance of the switch `-sv_lib` *pathname_without_ extension* (i.e., the filename shall be specified without the platform-specific extension). The SystemVerilog application is responsible for appending the appropriate extension for the actual platform. This switch can be used multiple times to define multiple libraries holding object code.

Both methods shall be provided and made available concurrently to permit any mixture of their usage. Every location can be an absolute pathname or a relative pathname, where the value of the switch `-sv_root` is used to identify an appropriate prefix for relative pathnames (see H.2 for more details on forming pathnames).

The following conditions also apply.

— The compiled object code itself shall be provided in the form of a shared library having the appropriate extension for the actual platform.

NOTE—Shared libraries use, for example, `.so` for Solaris and `.sl` for HP-UX; other operating systems might use different extensions. In any case, the SystemVerilog application needs to identify the appropriate extension.

— The provider of the compiled code is responsible for any external references specified within these objects. Appropriate data need to be provided to resolve all open dependencies with the correct information.

— The provider of the compiled code shall avoid interferences with other software and ensure the appropriate software version is taken (e.g., in cases where two versions of the same library are referenced). Similar problems can arise when there are dependencies on the expected run-time

environment in the compiled object code (e.g., in cases where C++ global objects or static initializers are used).

— The SystemVerilog application need only load object code within a shared library that is referenced by the SystemVerilog code or by registration functions; loading of additional functions included within a shared library can interfere with other parts.

In the case of multiple occurrences of the same file (files having the same pathname or that can easily be identified as being identical, e.g., by comparing the inodes of the files to detect cases where links are used to refer the same file), the above order also identifies the precedence of loading. A file located by method a (above in this subclause) shall override files specified by method b.

All compiled object code needs to be loaded in the specification order similarly to the above scheme; first the content of the bootstrap file is processed starting with the first line, then the set of `-sv_lib` switches is processed in order of their occurrence. Any library shall only be loaded once.

H.3.1 Bootstrap file

The object code bootstrap file has the following syntax:

a) The first line contains the string `#!SV_LIBRARIES`.

b) An arbitrary amount of entries follow, one entry per line, where every entry holds exactly one library location. Each entry consists only of the *pathname_without_extension* of the object code file to be loaded and can be surrounded by an arbitrary number of blanks; at least one blank shall precede the entry in the line. The value *pathname_without_extension* is equivalent to the value of the switch `-sv_lib`.

c) Any amount of comment lines can be interspersed between the entry lines; a comment line starts with the character # after an arbitrary (including zero) amount of blanks and is terminated with a newline.

H.3.2 Examples

a) If the pathname root has been set by the switch `-sv_root` to `/home/user` and the following object files need to be included:

```
/home/user/myclibs/lib1.so
/home/user/myclibs/lib3.so
/home/user/proj1/clibs/lib4.so
/home/user/proj3/clibs/lib2.so
```

then use either of the methods in Figure H-2. Both methods are equivalent.

```
#!SV_LIBRARIES
 myclibs/lib1
 myclibs/lib3
 proj1/clibs/lib4
 proj3/clibs/lib2
```
Bootstrap file method

```
...
-sv_lib myclibs/lib1
-sv_lib myclibs/lib3
-sv_lib proj1/clibs/lib4
-sv_lib proj3/clibs/lib2
...
```
Switch list method

Figure H-2—Using a simple bootstrap file or a switch list

b) If the current working directory is `/home/user`, using the series of switches shown in Figure H-3 (left column) results in loading the following files (right column):

```
-sv_lib svLibrary1
-sv_lib svLibrary2
-sv_root /home/project2/shared_code
-sv_lib svLibrary3
-sv_root /home/project3/code
-sv_lib svLibrary4
```

```
/home/user/svLibrary1.so
/home/user/svLibrary2.so

/home/project2/shared_code/svLibrary3.so

/home/project3/code/svLibrary4.so
```

Switches   **Files**

Figure H-3—Using a combination of `-sv_lib` **and** `-sv_root` **switches**

c) Further, using the set of switches and contents of bootstrap files shown in Figure H-4:

```
-sv_root /home/usr1
-sv_liblist bootstrap1

-sv_root /home/usr2
-sv_liblist /home/mine/bootstrap2
```

bootstrap1:

```
#! SV_LIBRARIES
  lib1
  lib2
```

bootstrap2:

```
#! SV_LIBRARIES
  lib3
  /common/libx
  lib5
```
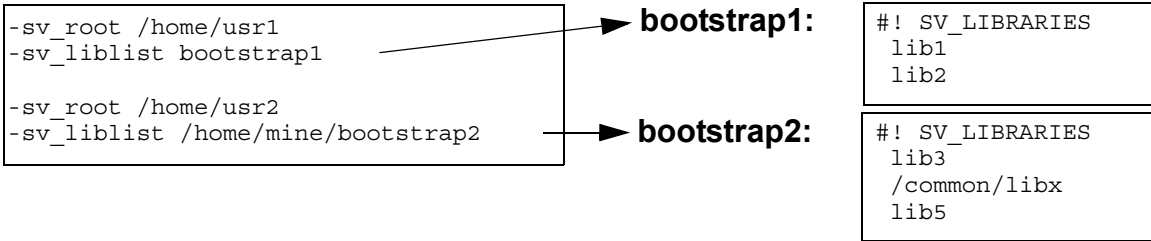
Figure H-4—Mixing `-sv_root` **and bootstrap files**

results in loading the following files:

```
/home/usr1/lib1.ext
/home/usr1/lib2.ext
/home/usr2/lib3.ext
/common/libx.ext
/home/usr2/lib5.ext
```

where *ext* stands for the actual extension of the corresponding file.

Annex I

(normative)

sv_vpi_user.h

```
/***************************************************************************
* sv_vpi_user.h
*
* SystemVerilog VPI extensions.
*
* This file contains the constant definitions, structure definitions, and
* routine declarations used by the Verilog PLI procedural interface VPI
* access routines.
*
***************************************************************************/

/***************************************************************************
* NOTE:
* The constant values 600 through 999 are reserved for use in this file.
* - the range 600-749 is reserved for SV VPI model extensions
* - the range 750-779 is reserved for the Coverage VPI
* - the range 800-899 is reserved for the Reader VPI
* Overlaps in the numerical ranges are permitted for different categories
*  of identifiers; e.g.
*  - object types
*  - properties
*  - callbacks
***************************************************************************/

#ifndef SV_VPI_USER_H
#define SV_VPI_USER_H

#include "vpi_user.h"

#ifdef __cplusplus
extern "C" {
#endif

/*************************** OBJECT TYPES ****************************/
#define vpiPackage              600
#define vpiInterface            601
#define vpiProgram              602
#define vpiInterfaceArray       603
#define vpiProgramArray         604
#define vpiTypespec             605
#define vpiModport              606
#define vpiInterfaceTfDecl      607
#define vpiRefObj               608

/* variables */
#define vpiVarBit               vpiRegBit
#define vpiLongIntVar           610
#define vpiShortIntVar          611
#define vpiIntVar               612
#define vpiShortRealVar         613
```

```
#define vpiByteVar              614
#define vpiClassVar             615
#define vpiStringVar            616
#define vpiEnumVar              617
#define vpiStructVar            618
#define vpiUnionVar             619
#define vpiBitVar               620
#define vpiLogicVar             vpiReg
#define vpiArrayVar             vpiRegArray
#define vpiClassObj             621

/* typespecs */
#define vpiLongIntTypespec      625
#define vpiShortRealTypespec    626
#define vpiByteTypespec         627
#define vpiShortIntTypespec     628
#define vpiIntTypespec          629
#define vpiClassTypespec        630
#define vpiStringTypespec       631
#define vpiEnumTypespec         633
#define vpiEnumConst            634
#define vpiIntegerTypespec      635
#define vpiTimeTypespec         636
#define vpiRealTypespec         637
#define vpiStructTypespec       638
#define vpiUnionTypespec        639
#define vpiBitTypespec          640
#define vpiLogicTypespec        641
#define vpiArrayTypespec        642
#define vpiVoidTypespec         643
#define vpiTypespecMember       644

#define vpiClockingBlock        650
#define vpiClockingIODecl       651
#define vpiClassDefn            652
#define vpiConstraint           653
#define vpiConstraintOrdering   654

#define vpiDistItem             645
#define vpiAliasStmt            646
#define vpiThread               647
#define vpiMethodFuncCall       648
#define vpiMethodTaskCall       649

/* concurrent assertions */
#define vpiAssert               686
#define vpiAssume               687
#define vpiCover                688

#define vpiDisableCondition     689
#define vpiClockingEvent        690
/* property decl, spec */
#define vpiPropertyDecl         655
#define vpiPropertySpec         656
#define vpiPropertyExpr         657
#define vpiMulticlockSequenceExpr 658
#define vpiClockedSeq           659
#define vpiPropertyInst         660
#define vpiSequenceDecl         661
```

```
#define vpiActualArgExpr          663
#define vpiSequenceInst           664
#define vpiImmediateAssert        665
#define vpiReturn                 666
/* pattern */
#define vpiAnyPattern             667
#define vpiTaggedPattern          668
#define vpiStructPattern          669
/* do .. while */
#define vpiDoWhile                670
/* waits */
#define vpiOrderedWait            671
#define vpiWaitFork               672
/* disables */
#define vpiDisableFork            673
#define vpiExpectStmt             674
#define vpiForeachStmt            675
#define vpiFinal                  676
#define vpiExtends                677
#define vpiDistribution           678
#define vpiIdentifier             679
#define vpiArrayNet               vpiNetArray
#define vpiEnumNet                680
#define vpiIntegerNet             681
#define vpiLogicNet               vpiNet
#define vpiTimeNet                682
#define vpiStructNet              683
#define vpiBreak                  684
#define vpiContinue               685


/******************************* METHODS *********************************/
/************* methods used to traverse 1 to 1 relationships **************/
#define vpiActual                 700

#define vpiTypedefAlias           701

#define vpiIndexTypespec          702
#define vpiBaseTypespec           703
#define vpiElemTypespec           704

#define vpiInputSkew              706
#define vpiOutputSkew             707
#define vpiDefaultClocking        709

#define vpiOrigin                 713
#define vpiPrefix                 714
#define vpiWith                   715

#define vpiProperty               718

#define vpiValueRange             720
#define vpiPattern                721
#define vpiWeight                 722

/************ methods used to traverse 1 to many relationships ************/
#define vpiTypedef                725
#define vpiImport                 726
#define vpiDerivedClasses         727
```

```
#define vpiMethods              730
#define vpiSolveBefore          731
#define vpiSolveAfter           732

#define vpiWaitingProcesses     734

#define vpiMessages             735
#define vpiLoopVars             737

#define vpiConcurrentAssertions 740
#define vpiMatchItem            741
#define vpiMember               742

/************* methods used to traverse 1 to many relationships **************/
#define vpiAssertion            744

/*********** methods used to traverse both 1-1 and 1-many relations ***********/
#define vpiInstance             745


/**************************************************************************/
/********************** generic object properties **********************/
/**************************************************************************/

#define vpiTop                  600

#define vpiUnit                 602

#define vpiAccessType           604
#define vpiForkJoinAcc      1
#define vpiExternAcc        2
#define vpiDPIExternAcc     3
#define vpiDPIImportAcc     4

#define vpiArrayType            606
#define vpiStaticArray      1
#define vpiDynamicArray     2
#define vpiAssocArray       3
#define vpiQueueArray       4
#define vpiArrayMember          607

#define vpiIsRandomized         608
#define vpiLocalVarDecls        609
#define vpiRandType             610
#define vpiNotRand          1
#define vpiRand             2
#define vpiRandC            3
#define vpiPortType             611
#define vpiInterfacePort    1
#define vpiModportPort      2
/* vpiPort is also a port type. It is defined in vpi_user.h */


#define vpiConstantVariable     612
#define vpiStructUnionMember    615

#define vpiVisibility           620
#define vpiPublicVis        1
```

```
#define vpiProtectedVis        2
#define vpiLocalVis            3


/* Return values for vpiConstType property */
#define vpiNullConst           8
#define vpiOneStepConst        9
#define vpiUnboundedConst      10


#define vpiAlwaysType            624
#define vpiAlwaysComb          2
#define vpiAlwaysFF            3
#define vpiAlwaysLatch         4


#define vpiDistType              625
#define vpiEqualDist             1 /* constraint equal distribution */
#define vpiDivDist               2 /* constraint divided distribution */


#define vpiPacked               630
#define vpiTagged               632
#define vpiRef                 6 /* Return value for vpiDirection property */
#define vpiVirtual              635
#define vpiIsConstraintEnabled  638


#define vpiClassType            640
#define vpiMailboxClass        1
#define vpiSemaphoreClass      2
#define vpiUserDefinedClass    3


#define vpiMethod               645
#define vpiIsClockInferred      649


#define vpiQualifier            650
#define vpiNoQualifier         0
#define vpiUniqueQualifier     1
#define vpiPriorityQualifier   2
#define vpiTaggedQualifier     4
#define vpiRandQualifier       8
#define vpiInsideQualifier     16


#define vpiInputEdge             651 /* returns vpiNoEdge, vpiPosedge,
vpiNegedge */
#define vpiOutputEdge            652 /* returns vpiNoEdge, vpiPosedge,
vpiNegedge */



/****************************** Operators ******************************/
#define vpiImplyOp             50 /* -> implication operator */
#define vpiNonOverlapImplyOp   51 /* |=> nonoverlapped implication */
#define vpiOverlapImplyOp      52 /* |-> overlapped implication operator */
#define vpiUnaryCycleDelayOp   53 /* binary cycle delay (##) operator */
#define vpiCycleDelayOp        54 /* binary cycle delay (##) operator */
#define vpiIntersectOp         55 /* intersection operator */
#define vpiFirstMatchOp        56 /* first_match operator */
#define vpiThroughoutOp        57 /* throught operator */
#define vpiWithinOp            58 /* within operator */
#define vpiRepeatOp            59 /* [=] nonconsecutive repetition */
#define vpiConsecutiveRepeatOp 60 /* [*] consecutive repetition */
#define vpiGotoRepeatOp        61 /* [->] goto repetition */
```

```
#define vpiPostIncOp            62 /* ++ post-increment */
#define vpiPreIncOp             63 /* ++ pre-increment */
#define vpiPostDecOp            64 /* -- post-decrement */
#define vpiPreDecOp             65 /* -- pre-decrement */

#define vpiMatchOp              66 /* match() operator */
#define vpiCastOp               67 /* type'() operator */
#define vpiIffOp                68 /* iff operator */
#define vpiWildEqOp             69 /* ==? operator */
#define vpiWildNeqOp            70 /* !=? operator */

#define vpiStreamLROp           71 /* left-to-right streaming {>>} operator */
#define vpiStreamRLOp           72 /* right-to-left streaming {<<} operator */

#define vpiMatchedOp            73 /* the .matched sequence operation */
#define vpiEndedOp              74 /* the .ended sequence operation */
#define vpiAssignmentPatternOp 75 /* '{} assignment pattern */
#define vpiMultiAssignmentPatternOp 76 /* '{n{}} multi assignment pattern */
#define vpiIfOp                 77 /* if operator */
#define vpiIfElseOp             78 /* if...else operator */
#define vpiCompAndOp            79 /* Composite and operator */
#define vpiCompOrOp             80 /* Composite or operator */
#define vpiTypeOp               81 /* type operator */

/********************** task/function properties **********************/
#define vpiOtherFunc            6 /* returns other types; for property vpiFuncType
*/

/********************** value for vpiValid **************************/
#define vpiValidUnknown         2 /* Validity of variable is unknown */

/*********************** STRUCTURE DEFINITIONS ***********************/

/************************* structure *************************/

/************************* CALLBACK REASONS *************************/
#define cbStartOfThread         600 /* callback on thread creation */
#define cbEndOfThread           601 /* callback on thread termination */
#define cbEnterThread           602 /* callback on reentering thread */
#define cbStartOfFrame          603 /* callback on frame creation */
#define cbEndOfFrame            604 /* callback on frame exit */
#define cbTypeChange            605 /* callback on variable type/size change */

/*********************** FUNCTION DECLARATIONS ***********************/


/*****************************************************************/
/************************ Coverage VPI ***************************/
/*****************************************************************/

/* coverage control */
#define vpiCoverageStart        750
#define vpiCoverageStop         751
#define vpiCoverageReset        752
#define vpiCoverageCheck        753
#define vpiCoverageMerge        754
#define vpiCoverageSave         755

/* coverage type properties */
```

```
#define vpiAssertCoverage          760
#define vpiFsmStateCoverage        761
#define vpiStatementCoverage       762
#define vpiToggleCoverage          763

/* coverage status properties */
#define vpiCovered                 765
#define vpiCoverMax                766
#define vpiCoveredCount            767
/* assertion-specific coverage status properties */
#define vpiAssertAttemptCovered    770
#define vpiAssertSuccessCovered    771
#define vpiAssertFailureCovered    772
/* FSM-specific coverage status properties */
#define vpiFsmStates               775
#define vpiFsmStateExpression      776

/* FSM handle types */
#define vpiFsm                     758
#define vpiFsmHandle               759


/***************************************************************************/
/*************************** Assertion VPI *******************************/
/***************************************************************************/

/* assertion callback types */
#define cbAssertionStart           606
#define cbAssertionSuccess         607
#define cbAssertionFailure         608
#define cbAssertionStepSuccess     609
#define cbAssertionStepFailure     610
#define cbAssertionDisable         611
#define cbAssertionEnable          612
#define cbAssertionReset           613
#define cbAssertionKill            614
/* assertion "system" callback types */
#define cbAssertionSysInitialized  615
#define cbAssertionSysOn           616
#define cbAssertionSysOff          617
#define cbAssertionSysKill         631
#define cbAssertionSysEnd          618
#define cbAssertionSysReset        619

/* assertion control constants */
#define vpiAssertionDisable        620
#define vpiAssertionEnable         621
#define vpiAssertionReset          622
#define vpiAssertionKill           623
#define vpiAssertionEnableStep     624
#define vpiAssertionDisableStep    625
#define vpiAssertionClockSteps     626
#define vpiAssertionSysOn          627
#define vpiAssertionSysOff         628
#define vpiAssertionSysKill        632
#define vpiAssertionSysEnd         629
#define vpiAssertionSysReset       630

typedef struct t_vpi_assertion_step_info {
```

```
   PLI_INT32 matched_expression_count;
   vpiHandle *matched_exprs;                    /* array of expressions */
PLI_INT32 stateFrom, stateTo;             /* identify transition */
} s_vpi_assertion_step_info, *p_vpi_assertion_step_info;

typedef struct t_vpi_attempt_info {
   union {
      vpiHandle failExpr;
      p_vpi_assertion_step_info step;
   } detail;
   s_vpi_time attemptStartTime;     /* Time attempt triggered */
} s_vpi_attempt_info, *p_vpi_attempt_info;

/* typedef for vpi_register_assertion_cb callback function */
typedef PLI_INT32(vpi_assertion_callback_func)(
   PLI_INT32 reason,                     /* callback reason */
   p_vpi_time cb_time,                   /* callback time */
   vpiHandle assertion,                  /* handle to assertion */
   p_vpi_attempt_info info,              /* attempt related information */
   PLI_BYTE8 *user_data                  /* user data entered upon registration */
);

vpiHandle vpi_register_assertion_cb(
   vpiHandle assertion,                  /* handle to assertion */
   PLI_INT32 reason,                     /* reason for which callbacks needed */
   vpi_assertion_callback_func *cb_rtn,
   PLI_BYTE8 *user_data                  /* user data to be supplied to cb */
);


/***************************************************************************/
/***************************** Reader VPI **********************************/
/***************************************************************************/


/********** Reader types ***********/
#define vpiTrvsObj              800 /* Data traverse object */
#define vpiCollection           810 /* Collection of VPI handle */
#define vpiObjCollection        811 /* Collection of traversable design objs */
#define vpiTrvsCollection       812 /* Collection of vpiTrvsObjs */

/********* Reader methods *********/

/* Check */
#define vpiIsLoaded             820 /* Object data are loaded check */
#define vpiHasDataVC            821 /* Traverse object has at least one VC
                                     * at some point in time in the
                                     * database check */

#define vpiHasVC                822 /* Has VC at specific time check */
#define vpiHasNoValue           823 /* Has no value at specific time check */
#define vpiBelong               824 /* Belongs to extension check */

/* Access */
#define vpiAccessLimitedInteractive 830 /* Interactive access */
#define vpiAccessInteractive        831 /* interactive with history access */
#define vpiAccessPostProcess        832 /* Database access */

/* Iteration on instances for loaded */
```

```
#define vpiDataLoaded           850 /* Use in vpi_iterate() */

/* Control Traverse/Check Time */
#define vpiMinTime              860 /* Min time */
#define vpiMaxTime              864 /* Max time */
#define vpiPrevVC               868 /* Previous Value Change (VC) */
#define vpiNextVC               870 /* Next Value Change (VC) */
#define vpiTime                 874 /* Time jump */


/********** Reader routines **********/

/* load extension form for the reader extension */
PLI_INT32 vpi_load_extension(PLI_BYTE8 *extension_name,
                             PLI_BYTE8 *name,
                             PLI_INT32 mode,
                             ...);

PLI_INT32 vpi_close(PLI_INT32 tool,
                    PLI_INT32 prop,
                    PLI_BYTE8* name);

PLI_INT32 vpi_load_init(vpiHandle objCollection,
                        vpiHandle scope,
                        PLI_INT32 level);

PLI_INT32 vpi_load(vpiHandle h);

PLI_INT32 vpi_unload(vpiHandle h);

vpiHandle vpi_create(PLI_INT32 prop,
                     vpiHandle h,
                     vpiHandle obj);

vpiHandle vpi_goto(PLI_INT32 prop,
                   vpiHandle obj,
                   p_vpi_time time_p,
                   PLI_INT32 *ret_code);

vpiHandle vpi_filter(vpiHandle h,
                     PLI_INT32 ft,
                     PLI_INT32 flag);


#ifdef __cplusplus
}
#endif

#endif
```

Annex J

(informative)

Glossary

For the purposes of this standard, the following terms and definitions apply. Other terms within IEEE standards are found in *The Authoritative Dictionary of IEEE Standards Terms* [B2].

aggregate: A set or collection of singular values, e.g., an aggregate expression, data object, or data type. An aggregate data type is any unpacked structure, unpacked union, or unpacked array data type. Aggregates may be copied or compared as a whole, but not typically used in an expression as a whole.

assertion: A statement that a certain property must be true, e.g., that a `read_request` must always be followed by a `read_grant` within two clock cycles. Assertions allow for automated checking that the specified property is true and can generate automatic error messages if the property is not true.

NOTE—SystemVerilog provides special assertion constructs, which are discussed in Clause 17.

bit-stream data type: Any data type whose values can be represented as a serial stream of bits. To qualify as a bit-stream data type, each and every bit of the values must be individually addressable. In other words, a bit-stream data type can be any data type except for a `handle`, `chandle`, `real`, `shortreal`, or `event`.

canonical representation: A data representation format established by convention into which and from which translations can be made with specialized representations.

constant: Either of two types of constants in SystemVerilog: elaboration constant or run-time constant. Parameters and local parameters are elaboration constants. Their values are calculated before elaboration is complete. Elaboration constants can be used to set the range of array types. Run-time constants are variables that can only be set in an initialization expression using the `const` qualifier.

context imported task: A direct programming interface (DPI) imported task declared with the 'context' property that is capable of calling exported tasks or functions and capable of accessing SystemVerilog objects via Verilog programming interface (VPI) or programming language interface (PLI) calls.

data object: A named entity that has a data value associated with it. Examples of data objects are nets, variables, and parameters. A data object has a data type that determines which values the data object can have.

data type: A set of values and a set of operations that can be performed on those values. Examples of data types are `logic`, `real`, and `string`. Data types can be used to declare data objects or to define user-defined data types that are constructed from other data types.

direct programming interface (DPI): An interface between SystemVerilog and foreign programming languages permitting direct function calls from SystemVerilog to foreign code and from foreign code to SystemVerilog. It has been designed to have low inherent overhead and permit direct exchange of data between SystemVerilog and foreign code.

disable protocol: A set of conventions for setting, checking, and handling disable status.

dynamic: Having values that can be resized or reallocated at run time. Dynamic arrays, associative arrays, queues, class handles, and data types that include such data types are dynamic data types.

elaboration: The process of binding together the components that make up a design. These components can include module instances, primitive instances, interfaces, and the top level of the design hierarchy.

enumerated type: Data types that can declare a data object that can have one of a set of named values. The numerical equivalents of these values can be specified. Values of an enumerated data type can be easily referenced or displayed using the enumerated names, as opposed to the enumerated values.

NOTE—See 4.10 for a discussion of enumerated types.

exported task: A SystemVerilog task that is declared in an export declaration and can be enabled from an imported task.

imported task: A direct programming interface (DPI) foreign code subprogram that can call exported tasks and can directly or indirectly consume simulation time.

integral: (A) A data type representing integer values. **(B)** A integer value that may be signed or unsigned, sliced into smaller integral values, or concatenated into larger values. *Syn:* vectored value. **(C)** An expression of an integral data type. **(D)** An object of an integral data type.

interface: An encapsulation of the communication between blocks of a design, allowing a smooth migration from abstract system-level design through successive refinement down to lower level register transfer and structural views of the design. By encapsulating the communication between blocks, the interface construct also facilitates design reuse. The inclusion of interface capabilities is one of the major advantages of SystemVerilog.

NOTE—Interfaces are covered in Clause 20.

language reference manual (LRM): "SystemVerilog LRM" refers to this standard. "Verilog LRM" refers to IEEE Std 1364 for Verilog hardware description language (HDL).

NOTE—See Clause 2 for information about IEEE Std 1364.

open array: A direct programming interface (DPI) array formal argument for which the packed or unpacked dimension size (or both) is not specified and for which interface routines describe the size of corresponding actual arguments at run time.

packed array: An array where the dimensions are declared before an object name. Packed arrays can have any number of dimensions. A one-dimensional packed array is the same as a vector width declaration in Verilog. Packed arrays provide a mechanism for subdividing a vector into subfields, which can be conveniently accessed as array elements. A packed array differs from an unpacked array, in that the whole array is treated as a single vector for arithmetic operations.

NOTE—Packed arrays are discussed in detail in Clause 5.

process: A thread of one or more programming statements that can be executed independently of other programming statements. Each **initial** procedure, **always** procedure, and continuous assignment statement in Verilog is a separate process. These are static processes. In other words, each time the process starts running, there is an end to the process. SystemVerilog adds specialized **always** procedures, which are also static processes, and dynamic processes. When dynamic processes are started, they can run without ending.

NOTE—Processes are presented in Clause 11.

signal: An informal term, usually meaning either a variable or net. The context where it is used may imply further restrictions on allowed types.

singular: An expression, data object, or data type that represents a single value, symbol, or handle. A singular data type is any data type except an unpacked structure, unpacked union, or unpacked array data type.

SystemVerilog: The IEEE 1800 set of abstract modeling and verification extensions to IEEE Std 1364. The many features of SystemVerilog are presented in this standard.

unpacked array: An array where the dimensions are declared after an object name. Unpacked arrays are the same as arrays in Verilog and can have any number of dimensions. An unpacked array differs from a packed array in that the whole array cannot be used for arithmetic operations. Each element must be treated separately.

NOTE—Unpacked arrays are discussed in Clause 5.

Verilog: The hardware description language (HDL) in IEEE Std 1364.

NOTE—See Clause 2 for information about IEEE Std 1364.

Verilog procedural interface (VPI): The third generation Verilog programming language interface (PLI), providing object-oriented access to Verilog behavioral, structural, assertion, and coverage objects.

Annex K

(informative)

Bibliography

[B1] *SystemVerilog 3.1a Language Reference Manual, Accellera's Extensions to Verilog*. 2004. Accellera, www.accellera.org.

[B2] IEEE 100, *The Authoritative Dictionary of IEEE Standards Terms*, Seventh Edition.

[B3] IEEE Std 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog™ Hardware Description Language.

[B4] IEEE Std 1364-2001, IEEE Standard Verilog® Hardware Description Language.

[B5] ISO/IEC 9899:1999, Programming Languages — C.

Index

Symbols

!=? wild inequality operator 102
expression operator 242
#1step 122
$, as unbounded range 67−69, 380
$assertkill 384
$assertoff 384
$asserton 384
$bits 38, 380
$bitstoshortreal 38
$cast 38
$cast() 90
$countones 264
$coverage_control 476
$coverage_merge 478
$coverage_save 478, 484
$dimensions 46, 382
$error 232, 383
$exit 229
$fatal 232, 383
$fell 246
$get_coverage 324
$high 46, 381
$increment 46, 381
$info 232, 383
$isunbounded 380
$isunknown 264, 384
$left 46, 381
$load_coverage_db 324
$low 46, 381
$onehot 264, 384
$onehot0 264, 384
$past 246
$readmemb 387, 389
$readmemh 387, 389
$right 46, 381
$root 333, 347
$rose 246
$sampled 246
$set_coverage_db_name 324
$shortrealtobits 38
$size 46, 382
$stable 246
$typename 379
$unit 332
$urandom 189
$urandom_range 190
$warning 232, 383
$writememb 387, 389
$writememh 387, 389
%= assignment operator 101

&= assignment operator 101
* expression operator 242
*= assignment operator 101−102
*= expression operator 242
*-> expression operator 242
+= assignment operator 101−102
.* port connections 342
.name port connections 342
/= assignment operator 101−102
:: scope resolution operator 93, 104, 189
:= weight operator 196
<<<= assignment operator 101
<<= assignment operator 101
-= assignment operator 101−102
==? wild equality operator 102
-> 171
>>= assignment operator 101
>>>= assignment operator 101
?: conditional operator 118
[* consecutive repetition operator 243
[= non-consecutive repetition operator 243, 245
[-> goto repetition operator 243, 245
\ line continuation 391
\a bell 12
\f form feed 12
\v vertical tab 12
\x02 hex number 12
^= assignment operator 101
`` double back tick 391
`cast operator 37
`define 391
`SV_COV_CHECK 475
`SV_COV_ERROR 475
`SV_COV_HIER 475
`SV_COV_NOCOV 475
`SV_COV_OK 475
`SV_COV_PARTIAL 475
`SV_COV_RESET 474
`SV_COV_START 474
`SV_COV_STOP 474
`SV_MODULE_COV 475
|= assignment operator 101
|=> implication operator 266, 268
|-> implication operator 266, 268−271

Numerics

2-state types 17
4-state types 17

A

access
    built-in packages 104
    class data hiding 91
    class members 83

E

efficiency 495
elaboration 632
empty() 558
encapsulation 91
enum 26
enumerated type methods
    first() 30
    last() 30
    name() 30
    next() 30
    num() 30
    prev() 30
enumerated types 26, 632
eq() 557
equivalent data types 77
erase() 560
erase_range() 560
exists() 54
expect 299
export 161, 362, 404, 410, 575, 578
extends 90
extern 94, 362, 366

F

filter 488, 493
final 127, 138
find() 60
find_first() 60
find_first_index() 60
find_index() 60
find_last() 60
find_last_index() 60
finish() 559
FINISHED process status 150
first() 30, 54
first_match 254
float 18
flush database 493
for loops 136
force 127
foreach 173−174
fork...join 147
fork…join_any 72
fork…join_none 72
forkjoin 349, 362, 366
form feed 12
front() 558
functions
    argument binding by name 160
    declaration syntax 155
    default arguments 159
    exporting 161, 362, 404, 410, 575, 578, 583
    importing 161, 362, 405

    in interfaces 362
    void 156

G

garbage collection 84
get_randstate() 190
getc() 22
goto 138
goto repetition operator 243, 245

H

handle 83
hextoa() 23
hierarchical names 347

I

icompare() 22
if statements, pattern matching 134
if...else 171−172, 197
iff 141, 234, 308, 315
ignore_bins 314, 318
illegal_bins 314, 318
immediate assertions 231
implication 171, 266, 268, 270−271
import 161, 362, 405
Inactive region 122
increment operator 99
inheritance 88, 169
initialize load access 493
initialize read interface 492
insert() 58, 559
insert_range() 560
inside operator 118
int 15, 17−18
integer 17−18
integer literals 12
integral 17
interactive access 495, 498
interface 349−375, 632
interfaces, restricting access 355
interfaces, virtual 370−375
interleaving 198
interoperability 485
intersect 242, 251, 316
iterative constraints 173
itoa() 23

J

join_any 145, 147
join_none 145, 147

K

kill() 150
KILLED process status 150

struct var 435
sys func call 444
sys task call 444
tagged pattern 460
task 443
task call 444
task func 443
tf call 444, 457
thread 446
time net 427
time typespec 434
time var 430
traversable 490
trvs obj 490
typespec 434
typespec member 434
union typespec 434
union var 435
var bit 430
var select 433
variable drivers 433
variable loads 433
variables 430, 433
void typespec 434
vpiMember 435
wait 459
wait fork 459
waits 457, 459
while 457
VPI read initialization 493
vpi_close() 492, 507
vpi_control() 471, 484
vpi_create() 492, 495–496, 508
vpi_filter() 488, 493, 502, 509
vpi_free_object() 492
vpi_get() 486, 492, 494
vpi_get_time() 487, 492, 497–498, 506
vpi_get_value() 486–487, 489, 492
vpi_get_vlog_info() 491
vpi_goto() 493, 497, 499, 508
vpi_handle() 492
vpi_handle_by_name() 495
vpi_iterate() 491
vpi_load() 489, 493, 495, 508
vpi_load_extension() 492, 495, 503–505, 507
vpi_load_init() 493–495, 497, 508
vpi_register_assertion_cb() 469
vpi_register_cb() 468
vpi_unload() 493, 503, 508

W

wait fork 145, 149
WAITING process status 150
waveform tool 485, 494

while 127, 136
wildcard
as a keyword 313
coverage bins 313
equality operators 102
index type 51
package import 329
pattern matching 131
with 59–60, 62, 183
within 242

X

xor() 62