Property Specification Language

Reference Manual

Version 1.1

June 9, 2004

Suggestions for improvements to the Property Specification Language and/or to this manual are welcome. They should be sent to the Property Specification Language email reflector

vfv@eda.org

The current Working Group's website address is

www.eda.org/vfv

The following individuals contributed to the creation, editing, and review of *Property Specification Language* version 1.0 and/or version 1.1.

Ken Albin	Motorola, Inc.	
Johan Alfredsson	Safelogic	
Thomas L. Anderson	0-In Design Automation, Inc.	
Roy Armoni	Intel, Corp.	
Shoham Ben-David	IBM Haifa Research Lab	
Jayaram Bhasker	Cadence Design Systems	
Kuang-Chien (KC) Chen	Verplex Systems, Inc.	
Edmund M. Clarke	Carnegie Mellon	
Ben Cohen	Consultant	
Simon Davidmann	Co-Design Automation, Inc	
Bernard Deadman	SDV, Inc	
Surrendra Dudani	Synopsys, Inc	
Cindy Eisner	IBM Haifa Research Lab	
E. Allen Emerson	University of Texas at Austin	
Andrea Fedeli	STMicroelectronics, Ltd.	
Dana Fisman	Weizmann Institute of Science, IBM Haifa Research Lab	
Tom Fitzpatrick	Co-Design Automation, Inc	
Limor Fix	Intel, Corp.	
Peter L. Flake	Co-Design Automation, Inc.	
Harry Foster	Jasper Design Automation, Inc., Verplex Systems, Inc.	Work Group Chair
Daniel Geist	IBM Haifa Research Lab	
Vassilios Gerousis	Infineon Technologies	
Michael J.C. Gordon	University of Cambridge	
John Havlicek	Motorola, Inc.	
Håkan Hjort	SafeLogic	
Richard Ho	0-In Design Automation, Inc.	
Yaron Kashai	Verisity Design, Inc.	
Joseph Lu	Nvidia, Inc., Sun Microsystems	
Adriana Maggiore	AltraVerifica Ltd., TransEDA Technology Ltd	
Erich Marschner	Cadence Design Systems	Work Group Co-Chair
Johan Mårtensson	Safelogic	
Anthony McIsaac	STMicroelectronics, Ltd.	
Hillel Miller	Motorola, Inc.	
Avigail Orni	IBM Haifa Research Lab	
Christian Pichler	Siemens	
Carl Pixley	Synopsys, Inc.	
Sitvanit Ruah	IBM Haifa Research Lab	
Ambar Sarkar	Paradigm Works	
Andrew Seawright	0-In Design Automation, Inc.	
Sandeep K. Shukla	University of California, Irvine	
Michael Siegel	Infineon Technologies	

Bassam Tabbara	Novas Software, Inc.
David Van Campenhout	Verisity Design, Inc.
Gal Vardi	Marvell Semiconductor, Ltd
Moshe Y. Vardi	Rice University
Bow-Yaw Wang	Verplex Systems, Inc.
Klaus Winkelmann	Infineon Technologies
Yaron Wolfsthal	IBM Haifa Research Lab

Revision history:

Version 0.1, 1st draft	05/10/02
Version 0.1, 2nd draft	05/17/02
Version 0.7, 1st draft	08/14/02
Version 0.7, 2nd draft	08/16/02
Version 0.7, 3rd draft	08/23/02
Version 0.7, 4th draft	08/26/02
Version 0.7, 5th draft	08/30/02
Version 0.7, 6th draft	09/08/02
Version 0.7, 7th draft	09/10/02
Version 0.8, 1st draft	09/12/02
Version 0.9, 1st draft	01/21/03
Version 0.95, 1st draft	01/26/03
Version 1.0	01/31/03
Version 1.01	04/25/03
Version 1.1, draft a	03/20/04
Version 1.1, draft b	04/17/04
Version 1.1, draft c	04/21/04
Version 1.1, draft d	04/21/04
Version 1.1, release	06/09/04

Table of Contents

1. Overview

1.1 Scope

This document specifies the syntax and semantics for the Accellera Property Specification Language.

1.2 Purpose

1.2.1 Motivation

Ensuring that a design's implementation satisfies its specification is the foundation of hardware verification. Key to the design and verification process is the act of specification. Yet historically, the process of specification has consisted of creating a natural language description of a set of design requirements. This form of specification is both ambiguous and, in many cases, unverifiable due to the lack of a standard machine-executable representation. Furthermore, ensuring that all functional aspects of the specification have been adequately *verified* (that is, covered) is problematic.

The Accellera Property Specification Language (PSL) was developed to address these shortcomings. It gives the design architect a standard means of specifying design properties using a concise syntax with clearly-defined formal semantics. Similarly, it enables the RTL implementer to capture design intent in a verifiable form, while enabling the verification engineer to validate that the implementation satisfies its specification through *dynamic* (that is, simulation) and *static* (that is, formal) verification means. Furthermore, it provides a means to measure the quality of the verification process through the creation of functional coverage models built on formally specified properties. Plus, it provides a standard means for hardware designers and verification engineers to rigorously document the design specification (machine-executable).

1.2.2 Goals

PSL was specifically developed to fulfill the following general hardware functional specification requirements:

— easy to learn, write, and read
— concise syntax
— rigorously well-defined formal semantics
— expressive power, permitting the specification for a large class of real world design properties
— known efficient underlying algorithms in simulation, as well as formal verification

1.3 Usage

PSL is a language for the formal specification of hardware. It is used to describe properties that are required to hold in the design under verification. PSL provides a means to write specifications that are both easy to read and mathematically precise. It is intended to be used for functional specification on the one hand and as input to functional verification tools on the other. Thus, a PSL specification is an executable documentation of a hardware design.

1.3.1 Functional specification

PSL can be used to capture requirements regarding the overall behavior of a design, as well as assumptions about the environment in which the design is expected to operate. PSL can also capture internal behavioral requirements and assumptions that arise during the design process. Both enable more effective functional verification and reuse of the design.

One important use of PSL is for documentation, either in place of or along with an English specification. A PSL specification can describe simple invariants (for example, signals `read_enable` and `write_enable` are never asserted simultaneously) as well as multi-cycle behavior (for example, correct behavior of an interface with respect to a bus protocol or correct behavior of pipelined operations).

A PSL specification consists of *assertions* regarding *properties* of a design under a set of *assumptions*. A *property* is built from *Boolean expressions*, which describe behavior over one cycle, *sequential expressions*, which describe multi-cycle behavior, and *temporal operators*, which describe relations over time between Boolean expressions and sequences. For example, consider the the following Verilog Boolean expression:

```
ena || enb
```

This expressions describes a cycle in which at least one of the signals `ena` and `enb` are asserted. The PSL sequential expression

```
{req; ack; !cancel}
```

describes a sequence of cycles, such that `req` is asserted in the first, `ack` in the second, and `cancel` deasserted in the third. They can be connected using the temporal operators `always` and `next` to get the property

```
always {req;ack;!cancel}(next[2] (ena || enb))
```

which means that following any sequence of `{req;ack;!cancel}` (i.e., `always`), either `ena` or `enb` is asserted two cycles later (i.e., `next[2]`). Adding the directive `assert` as follows:

```
assert always {req;ack;!cancel}(next[2] (ena || enb));
```

completes the specification, indicating that this property is expected to hold in the design and that this expectation needs to be verified.

1.3.2 Functional verification

PSL can also be used as input to verification tools, for both verification by simulation, as well as formal verification using a model checker or a theorem prover. Each of these is discussed below.

1.3.2.1 Simulation

A PSL specification can also be used to automatically generate checks of simulations. This can be done, for example, by directly integrating the checks in the simulation tool; by interpreting PSL properties in a testbench automation tool that drives the simulator; by generating HDL monitors that are simulated alongside the design; or by analyzing the traces produced at the end of the simulation.

For instance, the following PSL property:

```
always (req -> next !req)
```

states that signal `req` is a pulsed signal — if it is high in some cycle, then it is low in the following cycle. Such a property can be easily checked using a simulation checker written in some HDL that has the functionality of the Finite State Machine (FSM) shown in Figure 1.

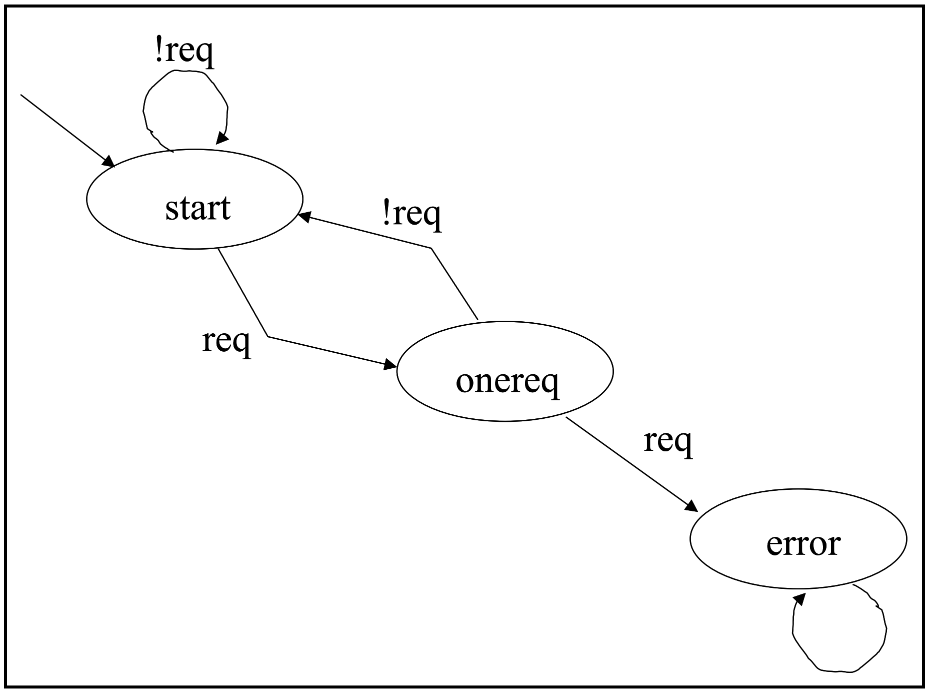

Figure 1—A simple (deterministic) FSM that checks the above property

For properties more complicated than the property shown above, manually writing a corresponding checker is painstaking and error-prone, and maintaining a collection of such checkers for a constantly changing design under development is a time-consuming task. Instead, a PSL specification can be used as input to a tool that automatically generates simulatable checkers.

Although in principle, all PSL properties can be checked for finite paths in simulation, the implementation of the checks is often significantly simpler for a subset called the *simple subset* of PSL. Informally, in this subset, composition of temporal properties is restricted to ensure that time *moves forward* from left to right through a property, as it does in a timing diagram. (See Section 4.4.4 for the formal definition of the simple subset.) For example, the property

```
always (a -> next[3](b))
```

which states that, if a is asserted, then b is asserted three cycles later, belongs to the simple subset, because a appears to the left of b in the property and also appears to the left of b in the timing diagram of any behavior that is not a violation of the property. Figure 2 shows an example of such a timing diagram.

An example of a property that is not in this subset is the property

```
always ((a && next[3](b)) -> c)
```

which states that, if a is asserted and b is asserted three cycles later, then c is asserted (in the same cycle as a). This property does not belong to the simple subset, because although c appears to the right of a and b in the property, it appears to the left of b in a timing diagram that is not a violation of the property. Figure 3 shows an example of such a timing diagram.

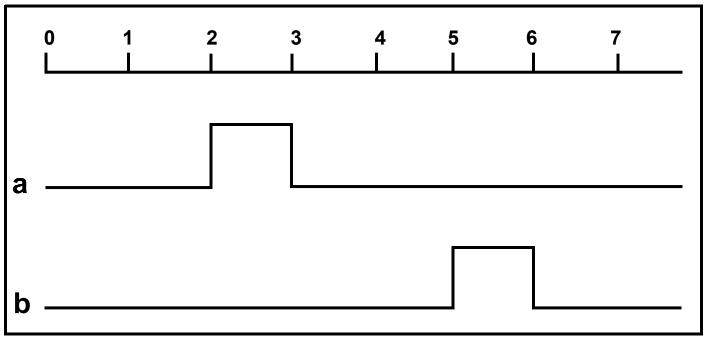

Figure 2—A trace that satisfies "always (a -> next[3] (b))"

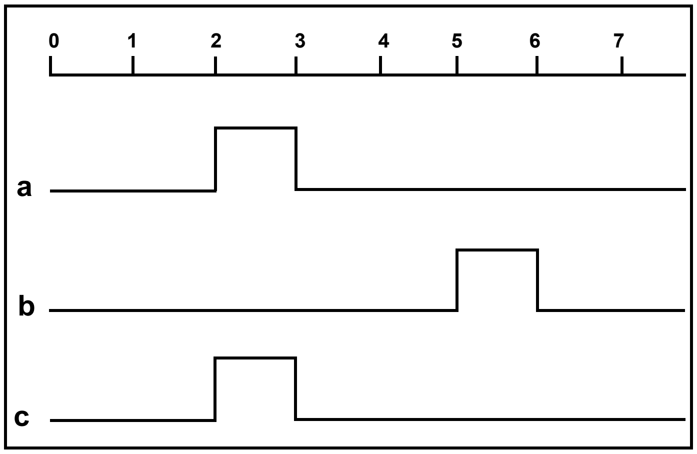

Figure 3—A trace that satisfies "always ((a && next[3] (b)) -> c)"

1.3.2.2 Formal verification

PSL is an extension of the standard temporal logics LTL and CTL. A specification in the PSL Foundation Language (respectively, the PSL Optional Branching Extension) can be *compiled down* to a formula of pure LTL (respectively, CTL), possibly with some auxiliary HDL code, known as a *satellite*.

1.4 Contents of this standard

The organization of the remainder of this standard is

— Chapter 2 (References) provides references to other applicable standards that are assumed or required for PSL.
— Chapter 3 (Definitions) defines terms used throughout this standard.
— Chapter 4 (Organization) describes the overall organization of the standard.
— Chapter 5 (Boolean layer) defines the Boolean layer.
— Chapter 6 (Temporal layer) defines the temporal layer.
— Chapter 7 (Verification layer) defines the verification layer.
— Chapter 8 (Modeling layer) defines the modeling layer.
— Appendix A (Syntax rule summary) summarizes the PSL syntax rules.
— Appendix B (Formal syntax and semantics of the temporal layer) defines the formal syntax and semantics of the temporal layer.

2. References

This standard shall be used in conjunction with the following publications. When any of the following standards is superseded by an approved revision, the revision shall apply.

The IEEE Standard Dictionary of Electrical and Electronics Terms, Sixth Edition.

IEEE Std 1076-2002, IEEE Standard VHDL Language Reference Manual.

IEEE Std 1076.6-1999, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis.

IEEE Std 1364-2001, IEEE Standard for Verilog Hardware Description Language.

IEEE P1364.1 (Draft 2.2, April 26,2002), Draft Standard for Verilog Register Transfer Level Synthesis.

Accellera 3.1a SystemVerilog Language Reference Manual

3. Definitions

For the purposes of this standard, the following terms and definitions apply. The *IEEE Standard Dictionary of Electrical and Electronics Terms* [B1] should be referenced for terms not defined in this standard.

3.1 Terminology

This section defines the terms used in this standard.

3.1.1 **assertion:** A statement that a given property is required to hold and a directive to verification tools to verify that it does hold.

3.1.2 **assumption:** A statement that the design is constrained by the given property and a directive to verification tools to consider only paths on which the given property holds.

3.1.3 **behavior:** A path.

3.1.4 **Boolean:** A Boolean expression.

3.1.5 **Boolean expression:** An expression that yields a logical value.

3.1.6 **checker:** An auxiliary process (usually constructed as a finite state machine) that monitors simulation of a design and reports errors when asserted properties do not hold. A checker may be represented in the same HDL code as the design or in some other form that can be linked with a simulation of the design.

3.1.7 **completes:** A sequential expression (or property) completes at the last cycle of any design behavior described by that sequential expression (or property).

3.1.8 **computation path:** A succession of states of the design, such that the design can actually transition from each state on the path to its successor.

3.1.9 **constraint:** A condition (usually on the input signals) that limits the set of behaviors to be considered. A constraint may represent real requirements (e.g., clocking requirements) on the environment in which the design is used, or it may represent artificial limitations (e.g., mode settings) imposed in order to partition the verification task.

3.1.10 **count:** A number or range.

3.1.11 **coverage:** A measure of the occurrence of certain behavior during (typically dynamic) verification and, therefore, a measure of the completeness of the (dynamic) verification process.

3.1.12 **cycle:** An evaluation cycle.

3.1.13 **describes:** A Boolean expression, sequential expression, or property describes the set of behaviors for which the Boolean expression, sequential expression, or property holds.

3.1.14 **design:** A model of a piece of hardware, described in some hardware description language (HDL). A design typically involves a collection of inputs, outputs, state elements, and combinational functions that compute next state and outputs from current state and inputs.

3.1.15 **design behavior:** A computation path for a given design.

3.1.16 **dynamic verification:** A verification process in which a property is checked over individual, finite design behaviors that are typically obtained by dynamically exercising the design through a finite number of evaluation cycles. Generally, dynamic verification supports no inference about whether the property holds for a behavior over which the property has not yet been checked.

3.1.17 **evaluation:** The process of exercising a design by iteratively applying values to its inputs, computing its next state and output values, advancing time, and assigning to the state variables and outputs their next values.

3.1.18 **evaluation cycle:** One iteration of the evaluation process. At an evaluation cycle, the state of the design is recomputed (and may change).

3.1.19 **extension:** An extension of a path is a path that starts with precisely the succession of states in the given path.

3.1.20 **False:** An interpretation of certain values of certain data types in an HDL. In the SystemVerilog and Verilog flavors, the single bit value `1’b0, 1’bx, 1’bz are` interpreted as the logical value *False*. In the VHDL flavor, he values STD.Standard.Boolean’(False) and STD.Standard.Bit’(‘0’), as well as the values IEEE.std_logic_1164.std_logic’(‘0’), IEEE.std_logic_1164.std_logic’(‘X’), and IEEE.std_logic_1164.std_logic’(‘Z’) are all interpreted as the logical value *False*. In the GDL flavor, the Boolean value `'false'` and bit value `0B` are both interpreted as the logical value *False*.

3.1.21 **finite range:** A range with a finite high bound.

3.1.22 **formal verification:** A verification process in which analysis of a design and a property yields a logical inference about whether the property holds for all behaviors of the design. If a property is declared true by a formal verification tool, no simulation can show it to be false. If the property does not hold for all behaviors, then the formal verification process should provide a specific counterexample to the property, if possible.

3.1.23 **holds:** A term used to talk about the meaning of a Boolean expression, sequential expression or property. Loosely speaking, a Boolean expression, sequential expression, or property holds in the first cycle of a path iff the path exhibits the behavior described by the Boolean expression, sequential expression, or property. The definition of holds for each form of Boolean expression, sequential expression, or property is given in the appropriate subsection of Chapter 6.

3.1.24 **holds tightly:** A term used to talk about the meaning of a sequential expression (SERE). Sequential expressions are evaluated over finite paths (behavior). Loosely speaking, a sequential expression holds tightly along a finite path iff the path exhibits the behavior described by the sequential expression. The definition of holds tightly for each form of SERE is given in the appropriate subsection of Section 6.1.

3.1.25 **liveness property:** A property that specifies an eventuality that is unbounded in time. Loosely speaking, a liveness property claims that "something good" eventually happens. More formally, a liveness property is a property for which any finite path can be extended to a path satisfying the property. For example, the property "whenever signal req is asserted, signal ack is asserted some time in the future" is a liveness property.

3.1.26 **logic type:** An HDL data type that includes values that are interpreted as logical values. A logic type may include values that are not interpreted as logical values. Such a logic type usually represents a multi-valued logic.

3.1.27 **logical value:** A value in the set {*True, False*}.

3.1.28 **model checking:** A type of formal verification.

3.1.29 **monitor:** *See:* **checker**.

3.1.30 **number:** A non-negative integer value, and a statically computable expression yielding such a value.

3.1.31 **occurs**, **occurrence:** A Boolean expression is said to “occur” in a cycle if it holds in that cycle. For example, “the next occurrence of the Boolean expression” refers to the next cycle in which the Boolean expression holds.

3.1.32 **path:** A succession of states of the design, whether or not the design can actually transition from one state on the path to its successor.

3.1.33 **positive count:** A positive number or a positive range.

3.1.34 **positive number:** A number that is greater than zero (0).

3.1.35 **positive range:** A range with a low bound that is greater than zero (0).

3.1.36 **prefix:** A prefix of a given path is a path of which the given path is an extension.

3.1.37 **property:** A collection of logical and temporal relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behaviors.

3.1.38 **range:** A series of consecutive numbers, from a low bound to a high bound, inclusive, such that the low bound is less than or equal to the high bound. In particular, this includes the case in which the low bound is equal to the high bound. Also, a pair of statically computable integer expressions specifying such a series of consecutive numbers, where one expression specifies the low bound of the series, and the other expression specifies the high bound of the series. A range may describe time range, event repetitions or bits of a bus or a vector. For time and repetition range, the low bound must be the left integer and the high bound must be the right integer. For vectors and bus bits range the order is not important, unless restricted by the underlying flavor.

3.1.39 **required** (to hold)**:** A property is required to hold if the design is expected to exhibit behaviors that are within the set of behaviors described by the property.

3.1.40 **restriction:** A statement that the design is constrained by the given sequential expression and a directive to verification tools to consider only paths on which the given sequential expression holds.

3.1.41 **safety property:** A property that specifies an invariant over the states in a design. The invariant is not necessarily limited to a single cycle, but it is bounded in time. Loosely speaking, a safety property claims that "something bad" does not happen. More formally, a safety property is a property for which any path violating the property has a finite prefix such that every extension of the prefix violates the property. For example, the property, "whenever signal req is asserted, signal ack is asserted within 3 cycles" is a safety property.

3.1.42 **sequence:** A sequential expression that is enclosed in curly braces.

3.1.43 **sequential expression:** A finite series of terms that represent a set of behaviors.

3.1.44 **SERE:** A sequential expression.

3.1.45 **simulation:** A type of dynamic verification.

3.1.46 **starts:** A sequential expression starts at the first cycle of any behavior for which it holds. In addition, a sequential expression starts at the first cycle of any behavior that is the prefix of a behavior for which it holds. For example, if `a` holds at cycle 7 and `b` holds in every cycle from 8 onward, then the sequential expression `{a; b[*] ;c}` starts at cycle 7.

3.1.47 **strictly before:** Before, and not in the same cycle as.

3.1.48 **strong operator:** A temporal operator, the (non-negated) use of which creates a liveness property.

3.1.49 **terminating condition:** A Boolean expression, the occurrence of which causes a property to complete.

3.1.50 **terminating property:** A property that, when it holds, causes another property to complete.

3.1.51 **True:** An interpretation of certain values of certain data types in an HDL.
In the SystemVerilog and Verilog flavors, the single bit value `1'b1` is interpreted as the logical value *True*. In the VHDL flavor, the values `STD.Standard.Boolean'(True)`, `STD.Standard.Bit'('1')`, and `IEEE.std_logic_1164.std_logic'('1')` are all interpreted as the logical value *True*. In the GDL flavor, the Boolean value `'true'` and bit value `1B` are both interpreted as the logical value *True*.

3.1.52 **verification:** The process of confirming that, for a given design and a given set of constraints, a property that is required to hold in that design actually does hold under those constraints.

3.1.53 **weak operator:** A temporal operator, the (non-negated) use of which does not create a liveness property.

3.2 Acronyms and abbreviations

This section lists the acronyms and abbreviations used in this standard.

BNF	extended Backus-Naur Form
cpp	C pre-processor
CTL	computation tree logic
EDA	electronic design automation
FL	Foundation Language
FSM	finite state machine
GDL	General Description Language
HDL	hardware description language
iff	if and only if
LTL	linear-time temporal logic
PSL	Property Specification Language
OBE	Optional Branching Extension
RTL	Register Transfer Level
SERE	Sequential Extended Regular Expression
VHDL	VHSIC Hardware Description Language

4. Organization

4.1 Abstract structure

PSL consists of four layers, which cut the language along the axis of functionality. PSL also comes in four flavors, which cut the language along the axis of HDL compatibility. Each of these is explained in detail in the following sections.

4.1.1 Layers

PSL consists of four layers: Boolean, temporal, verification, and modeling.

4.1.1.1 Boolean layer

This layer is used to build expressions that are, in turn, used by the other layers. Although it contains expressions of many types, it is known as the *Boolean layer* because it is the *supplier* of Boolean expressions to the heart of the language — the temporal layer. Boolean layer expressions are evaluated in a single evaluation cycle.

4.1.1.2 Temporal layer

This layer is the heart of the language; it is used to describe properties of the design. It is known as the *temporal layer* because, in addition to simple properties, such as "signals `a` and `b` are mutually exclusive", it can also describe properties involving complex temporal relations between signals, such as, "if signal `c` is asserted, then signal `d` shall be asserted before signal `e` is asserted, but no more than eight clock cycles later." Temporal expressions are evaluated over a series of evaluation cycles.

4.1.1.3 Verification layer

This layer is used to tell the verification tools what to do with the properties described by the temporal layer. For example, the verification layer contains directives that tell a tool to verify that a property holds or to check that a specified sequence is covered by some test case.

4.1.1.4 Modeling layer

This layer is used to model the behavior of design inputs (for tools, such as formal verification tools, which do not use test cases) and to model auxiliary hardware that is not part of the design, but is needed for verification.

4.1.2 Flavors

PSL comes in four *flavors*: one for each of the hardware description languages SystemVerilog, Verilog, VHDL, and GDL[1]. The syntax of each flavor conforms to the syntax of the corresponding HDL in a number of specific areas — a given flavor of PSL is compatible with the corresponding HDL's syntax in those areas.

4.1.2.1 SystemVerilog flavor

In this flavor, all expressions of the Boolean layer, as well as modeling layer code, are written in SystemVerilog syntax (see Accellera SystemVerilog version 3.1a). The SystemVerilog flavor also has limited influence on the syntax of the temporal layer. For example, ranges of the temporal layer are specified using the SystemVerilog-style syntax `i:j`.

[1]The definition of GDL is not yet available publicly. This flavor is included in the LRM as a placeholder for future development.

4.1.2.2 Verilog flavor

In this flavor, all expressions of the Boolean layer, as well as modeling layer code, are written in Verilog syntax (see IEEE Std 1364-2001)[2]. The Verilog flavor also has limited influence on the syntax of the temporal layer. For example, ranges of the temporal layer are specified using the Verilog-style syntax `i:j`.

4.1.2.3 VHDL flavor

In this flavor, all expressions of the Boolean layer, as well as modeling layer code, are written in VHDL syntax. (See IEEE Std 1076-2002). The VHDL flavor also has some influence on the syntax of the temporal layer. For example, ranges of the temporal layer are specified using the VHDL-style syntax `i to j`.

4.1.2.4 GDL flavor

In this flavor, all expressions of the Boolean layer, as well as modeling layer code, are written in GDL syntax. The GDL flavor also has some influence on the syntax of the temporal layer. For example, ranges of the temporal layer are specified using the GDL-style syntax `i..j`.

4.2 Lexical structure

This section defines the identifiers, keywords, operators, macros and comments used in PSL.

4.2.1 Identifiers

Identifiers in PSL consist of an alphabetic character, followed by zero or more alphanumeric characters; each subsequent alphanumeric character may optionally be preceded by a single underscore character.

PSL identifiers are case-sensitive in the SystemVerilog and Verilog flavors and case-insensitive in the VHDL and GDL flavors.

Example

```
mutex
Read_Transaction
L_123
```

4.2.2 Keywords

Keywords are reserved identifiers in PSL, so an HDL name that is a PSL keyword cannot be referenced directly, by its simple name, in an HDL expression used in a PSL property. However, such an HDL name can be referenced indirectly, using a hierarchical name or qualified name as allowed by the underlying HDL.

[2]For more information on references, see Chapter 2.

The keywords used in PSL are shown in Table 1.

Table 1—Keywords

A	**E**	**next_a**	**sequence**
AF	**EF**	**next_a!**	**stable**
AG	**EG**	**next_e**	**strong**
AX	**EX**	**next_e!**	
abort	**endpoint**	**next_event**	**to**[e]
always	**eventually!**	**next_event!**	
		next_event_a	**U**
and[a]	**F**	**next_event_a!**	**union**
assert	**fairness**	**next_event_e**	**until**
assume	**fell**	**next_event_e!**	**until!**
assume_guarantee	**forall**	**not**[c]	**until!_**
			until_
before	**G**	**onehot**	
before!		**onehot0**	**vmode**
before!_	**in**	**or**[d]	**vprop**
before_	**inf**		**vunit**
boolean	**inherit**	**property**	
	is[b]		**W**
	isunknown	**prev**	**within**
clock			
const	**never**	**report**	**X**
countones	**next**	**restrict**	**X!**
cover	**next!**	**restrict_guarantee**	
		rose	
default			

[a]**and** is a keyword only in the VHDL flavor; see the flavor macro `AND_OP` (4.3.2).
[b]**is** is a keyword only in the VHDL flavor; see the flavor macro `DEF_SYM` (4.3.2).
[c]**not** is a keyword only in the VHDL flavor; see the flavor macro `NOT_OP` (4.3.2).
[d]**or** is a keyword only in the VHDL flavor; see the flavor macro `OR_OP` (4.3.2).
[e]**to** is a keyword only in the VHDL flavor; see the flavor macro `RANGE_SYM` (4.3.2).

4.2.3 Operators

4.2.3.1 HDL operators

For a given flavor of PSL, the operators of the underlying HDL have the highest precedence. In particular, this includes logical, relational, and arithmetic operators of the HDL. The HDL's logical operators for negation, conjunction, and disjunction of Boolean values can be used in PSL for negation, conjunction, and disjunction of properties as well. In such applications, those operators have their usual precedence and associativity, as if the PSL properties that are operands produced Boolean values of a type appropriate to the logical operators native to the HDL.

4.2.3.2 Foundation Language (FL) operators

Various operators are available in PSL. Each operator has a precedence relative to other operators. In general, operators with a higher relative precedence are associated with their operands before operators with a lower relative precedence. If two operators with the same precedence appear in sequence, then the operators are associated with their operands according to the associativity of the operators. Left-associative operators are associated with operands in left-to-right order of appearance in the text; right-associative operators are associated with operands in right-to-left order of appearance in the text.

Table 2—FL operator precedence and associativity

Operator class	Associativity	Operators	
(highest precedence) HDL operators			
Union operator	left	`union`	
Clocking operator	left	`@`	
SERE repetition operators	left	`[* ]` `[+]` `[= ]` `[-> ]`	
Sequence within operator	left	`within`	
Sequence AND operators	left	`&` `&&`	
Sequence OR operator	left	`|`	
Sequence fusion operator	left	`:`	
Sequence concatenation operator	left	`;`	
FL termination operator	left	`abort`	
FL occurrence operators	right	`next*` `X` `X!`	`eventually!` `F`
FL bounding operators	right	`U` `until*`	`W` `before*`
Sequence implication operators	right	`|->`	`|=>`
Boolean implication operators	right	`->`	`<->`
FL invariance operators *(lowest precedence)*	right	`always` `G`	`never`

NOTE—The notation `next*` represents the *next* family of operators, which includes the operators `next`, `next!`, `next_a`, `next_a!`, `next_e`, `next_e!`, `next_event`, `next_event!`, `next_event_a!`, and `next_event_e!`. The notation `until*` represents the *until* family of operators, which includes the operators `until`, `until!`, `until_`, and `until!_`. The notation `before*` represents the *before* family of operators, which includes the operators `before`, `before!`, `before_`, and `before!_`.

4.2.3.2.1 Clocking operator

For any flavor of PSL, the FL operator with the next highest precedence after the HDL operators is that used to specify the clock expression that controls when the property is evaluated. The following operator is the unique member of this class:

`@`	clock event

The clocking operator is left-associative.

4.2.3.2.2 SERE repetition operators

For any flavor of PSL, the Foundation Language (FL) operators with the next highest precedence are the repetition operators that construct Sequential Extended Regular Expressions (SEREs). These operators are:

`[* ]`	consecutive repetition
`[+]`	consecutive repetition
`[= ]`	non-consecutive repetition
`[-> ]`	goto repetition

SERE repetition operators are left-associative.

4.2.3.2.3 Sequence within operator

For any flavor of PSL, the FL operator with the next highest precedence is the sequence within operator, which is used to describe behavior in which one sequence occurs during the course of another, or within a time-bounded interval:

`within`	sequence within operator

The sequence within operator is left-associative.

4.2.3.2.4 Sequence conjunction operators

For any flavor of PSL, the FL operators with the next highest precedence are the sequence conjunction operators, which are used to describe behavior consisting of parallel paths. These operators are:

&	non-length-matching sequence conjunction
&&	length-matching sequence conjunction

Sequence conjunction operators are left-associative.

4.2.3.2.5 Sequence disjunction operator

For any flavor of PSL, the FL operator with the next highest precedence is the sequence disjunction operator, which is used to describe behavior consisting of alternative paths:

`|`	sequence disjunction

The sequence disjunction operator is left-associative.

4.2.3.2.6 Sequence fusion operator

For any flavor of PSL, the FL operator with the next highest precedence is the sequence fusion operator, which is used to describe behavior in which a later sequence starts in the same cycle in which an earlier sequence finishes:

`:`	sequence fusion

The sequence fusion operator is left-associative.

4.2.3.2.7 Sequence concatenation operator

For any flavor of PSL, the FL operator with the next highest precedence is the sequence concatenation operator, which is used to describe behavior in which one sequence is followed by another:

`;`	sequence concatenation

The sequence contatenation operator is left-associative.

4.2.3.2.8 FL termination operator

For any flavor of PSL, the FL operator with the next highest precedence is the FL termination operator, used to specify a condition which will cause both current and future obligations to be canceled:

`abort`	immediate termination of current and future obligations

The FL termination operator is left-associative.

4.2.3.2.9 FL occurrence operators

For any flavor of PSL, the FL operators with the next highest precedence are those used to specify when a subordinate property must hold, if the parent property is to hold. These operators are:

`eventually!`	must hold at some time in the indefinite future
`next*`	must hold at some specified future time or range of future times

FL occurrence operators are right-associative.

4.2.3.2.10 Bounding operators

For any flavor of PSL, the FL operators with the next highest precedence are those used to specify a condition after which the parent property need not hold. These operators are:

`until*`	must hold up to a given event
`before*`	must hold at some time before a given event

FL bounding operators are right-associative.

4.2.3.2.11 Suffix implication operators

For any flavor of PSL, the FL operators with the next highest precedence are those used to describe behavior consisting of a property that holds at the end of a given sequence. These operators are:

`|->`	overlapping suffix implication
`|=>`	non-overlapping suffix implication

The suffix implication operators are right-associative.

NOTE—The FL Property {r} (f) is an alternative form for (and has the same semantics as) the FL Property {r} |-> f.

4.2.3.2.12 Logical implication operators

For any flavor of PSL, the FL operators with the next highest precedence are those used to describe behavior consisting of a Boolean, a sequence, or a property that holds if another Boolean, sequence, or property holds. These operators are:

`->`	logical IF implication
`<->`	logical IFF implication

The logical IF and logical IFF implication operators are right-associative.

4.2.3.2.13 FL invariance operators

For any flavor of PSL, the FL operators with the next highest precedence are those used to specify when a subordinate property must hold, if the parent property is to hold. These operators are:

`always`	must hold, globally
`never`	must NOT hold, globally

FL occurrence operators are right-associative.

4.2.3.3 Optional Branching Extension (OBE) operators

Operator class	Associativity	Operators
(highest precedence) HDL operators		
OBE occurrence operators	left	`AX` `AG` `AF` `EX` `EG` `EF` `A [ U ]` `E [ U ]`
Boolean implication operators *(lowest precedence)*	right	`->` `<->`

Table 3—OBE operator precedence and associativity

4.2.3.3.1 OBE occurrence operators

For any flavor of PSL, the Optional Branching Extension (OBE) operators operators with the next highest precedence after the HDL operators after the HDL operators are those used to specify when a subordinate property must hold, if the parent property is to hold. These operators include the following:

`AX`	on all paths, at the next state on each path
`AG`	on all paths, at all states on each path
`AF`	on all paths, at some future state on each path
`EX`	on some path, at the next state on the path
`EG`	on some path, at all states on the path
`EF`	on some path, at some future state on the path
`A[ U ]`	on all paths, in every state up to a certain state on each path
`E[ U ]`	on some path, in every state up to a certain state on that path

The OBE occurrence operators are left-associative.

4.2.3.3.2 OBE implication operators

For any flavor of PSL, the OBE operators with the next highest precedence are those used to build properties from Boolean expressions and subordinate properties through implication. These operators include:

`->`	logical IF implication
`<->`	logical IFF implication

4.2.4 Macros

PSL provides macro processing capabilities that facilitate the definition of properties. VHDL and GDL flavors support cpp pre-processing directives (e.g., #define, #ifdef, #else, #include, and #undef). SystemVerilog and Verilog flavors support Verilog compiler directives (e.g., `define, `ifdef, `else, `include, and `undef). All flavors also support PSL macros %for and %if, which can be used to conditionally or iteratively generate PSL statements. The cpp style and Verilog compiler directives must be fully parsed first, the PSL macros must be parsed in the second iteration, and any underlying flavor directives must be parsed last.

4.2.4.1 The %for construct

The `%for` construct replicates a piece of text a number of times, with the possibility of each replication receiving a parameter. The syntax of the `%for` construct is as follows:

```
%for /var/ in /expr1/ .. /expr2/ do
    ...
%end
```

or:

```
%for /var/ in { /item/, /item/, ... , /item/ } do
    ...
%end
```

The replicator name *var* is any legal PSL identifier name. It cannot be the same as any other identifier (variable, unit name, design signal etc.) except another non wrapping PSL replicator var. The replication expressions *exprn* must be statically computed expressions resulting a legal PSL range. A replication item *item* is any legal PSL alphanumeric string or previously defined cpp style macro.

In the first case, the text inside the %for-%end pairs will be replicated expr2-expr1+1 times (assuming that expr2>=expr1). In the second case, the text will be replicated according to the number of items in the list. During each replication of the text, the loop variable value is substituted into the text as follows. Suppose the loop variable is called ii. Then the current value of the loop variable can be accessed from the loop body using the following three methods:

The current value of the loop variable can be accessed using simply ii if ii is a separate token in the text. For instance:

```
%for ii in 0..3 do
    define aa(ii) := ii > 2;
%end
```

is equivalent to:

```
define aa(0) := 0 > 2;
define aa(1) := 1 > 2;
define aa(2) := 2 > 2;
define aa(3) := 3 > 2;
```

If ii is part of an identifier, the value of ii can be accessed using %{ii} as follows:

```
%for ii in 0..3 do
    define aa%{ii} := ii > 2;
%end
```

is equivalent to:

```
define aa0 := 0 > 2;
define aa1 := 1 > 2;
define aa2 := 2 > 2;
define aa3 := 3 > 2;
```

If ii needs to be used as part of an expression, it can be accessed as follows:

```
%for ii in 1..4 do
    define aa%{ii-1} := %{ii-1} > 2;
%end
```

The above is equivalent to:

```
define aa0 := 0 > 2;
define aa1 := 1 > 2;
define aa2 := 2 > 2;
define aa3 := 3 > 2;
```

The following operators can be used in pre-processor expressions:

```
=                  !=
<                  >
<=                 >=
+                  -
*                  /
                   %
```

4.2.4.2 The %if construct

The `%if` construct is similar to the `#if` construct of the cpp pre-processor. However, `%if` must be used when it is conditioned on variables defined in an encapsulating `%for`. The syntax of `%if` is as follows:

```
%if /expr/ %then
    ....
%end
```

or:

```
%if /expr/ %then
    ...
%else
    ...
%end
```

4.2.5 Comments

PSL provides the ability to add comments to PSL specifications. For each flavor, the comment capability is consistent with that provided by the corresponding HDL environment.

For the SystemVerilog and Verilog flavors, both the block comment style (`/* .... */`) and the trailing comment style (`// ....` <*eol*>) are supported.

For the VHDL flavor, the trailing comment style (`-- ....` <*eol*>) is supported.

For the GDL flavor, both the block comment style (`/* .... */`) and the trailing comment style (`-- ....` <*eol*>) are supported.

4.3 Syntax

4.3.1 Conventions

The formal syntax described in this standard uses the following extended Backus-Naur Form (BNF).

a) The initial character of each word in a nonterminal is capitalized. For example:

    PSL_Statement

   A nonterminal can be either a single word or multiple words separated by underscores. When a multiple-word nonterminal containing underscores is referenced within the text (e.g., in a statement that describes the semantics of the corresponding syntax), the underscores are replaced with spaces.

b) Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. For example:

    vunit (;

c) The `::=` operator separates the two parts of a BNF syntax definition. The syntax category appears to the left of this operator and the syntax description appears to the right of the operator. For example, item d) shows three options for a *Vunit_Type*.

d) A vertical bar separates alternative items (use one only) unless it appears in boldface, in which case it stands for itself. For example:

    Vunit_Type ::= **vunit** | **vprop** | **vmode**

e) Square brackets enclose optional items unless they appear in boldface, in which case they stand for themselves. For example:

    Sequence_Declaration ::= **sequence** Name [**(** Formal_Parameter_List **)**] DEF_SYM Sequence **;**

   indicates (*Formal_Parameter_List*) is an optional syntax item for *Sequence_Declaration*, whereas

    | Sequence **[*** [Range] **]**

   indicates that (the outer) square brackets are part of the syntax, while Range is optional.

f) Braces enclose a repeated item unless they appear in boldface, in which case they stand for themselves. A repeated item may appear zero or more times; the repetition is equivalent to that given by a left-recursive rule. Thus, the following two rules are equivalent:

    Formal_Parameter_List ::= Formal_Parameter { **;** Formal_Parameter }
    Formal_Parameter_List ::= Formal_Parameter | Formal_Parameter_List **;** Formal_Parameter

g) A comment in a production is preceded by a colon (**:**) unless it appears in boldface, in which case it stands for itself.

h) 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, *vunit_*Name is equivalent to Name.

The main text uses *italicized* type when a term is being defined, and `monospace` font for examples and references to constants such as `0`, `1`, or `X` values.

4.3.2 HDL dependencies

PSL is defined in several flavors, each of which corresponds to a particular hardware description language with which PSL can be used. *Flavor macros* reflect the flavors of PSL in the syntax definition. A flavor macro is similar to a grammar production, in that it defines alternative replacements for a nonterminal in the grammar. A flavor macro is different from a grammar production, in that the alternatives are labeled with an HDL name and in the context of a given HDL, only the alternative labeled with that HDL name can be selected.

The name of each flavor macro is shown in all uppercase. Each flavor macro defines analogous, but possibly different syntax choices allowed for each flavor. The general format is the term `Flavor Macro`, then the actual *macro name*, followed by the = operator, and, finally, the definition for each of the HDLs.

Example

```
Flavor Macro RANGE_SYM =
        SystemVerilog: : / Verilog: : / VHDL: to / GDL: ..
```

shows the range symbol macro (RANGE_SYM).

PSL also defines a few extensions to Verilog declarations as shown in Box 1.

```
Extended_Verilog_Declaration ::=
    Verilog_module_or_generate_item_declaration
  | Extended_Verilog_Type_Declaration
```

Box 1—Extended Verilog Declarations

4.3.2.1 HDL_UNIT

At the topmost level, a PSL specification consists of a set of HDL design units and a set of PSL verification units. The Flavor Macro `HDL_UNIT` identifies the nonterminals that represent top-level design units in the grammar for each of the respective HDLs, as shown in Box 2.

```
Flavor Macro HDL_UNIT =
    SystemVerilog: SystemVerilog_module_declaration
  / Verilog: Verilog_module_declaration
  / VHDL: VHDL_design_unit
  / GDL: GDL_module_declaration
```

Box 2—Flavor macro HDL_UNIT

4.3.2.2 HDL_MOD_NAME

PSL verification units refer to HDL modules or design units. The Flavor Macro HDL_MOD_NAME specifies how modules and design units can be referred to in the various flavors.

```
Flavor Macro HDL_MOD_NAME =
        SystemVerilog: module_Name
        / Verilog: module_Name
        / VHDL: entity_aspect
        / GDL: module_Name
```

Box 3—Flavor macro HDL_MOD_NAME

4.3.2.3 HDL_DECL and HDL_STMT

PSL verification units may contain certain kinds of HDL declarations and statements. Flavor macros `HDL_DECL` and `HDL_STMT` connect the PSL syntax with the syntax for declarations and statements in the grammar for each HDL. Both of these are shown in Box 4.

```
Flavor Macro HDL_DECL  =
      SystemVerilog: SystemVerilog_module_or_generate_item_declaration
    / Verilog: Extended_Verilog_Declaration
    / VHDL: VHDL_declaration
    / GDL: GDL_module_item_declaration
Flavor Macro HDL_STMT  =
      SystemVerilog: SystemVerilog_module_or_generate_item
    / Verilog: Verilog_module_or_generate_item
    / VHDL: VHDL_concurrent_statement
    / GDL: GDL_module_item
```

Box 4—Flavor macros HDL_DECL and HDL_STMT

4.3.2.4 HDL_EXPR

Expressions shall be valid expressions in the underlying HDL description. This applies to expressions appearing directly within a temporal layer property, as well as to any sub-expressions of those expressions. The definition of `HDL_EXPR` captures this requirement, as shown in Box 5.

```
Flavor Macro HDL_EXPR =
      SystemVerilog: SystemVerilog_Expression
    / Verilog: Verilog_Expression
    / VHDL: VHDL_Expression
    / GDL: GDL_Expression
```

Box 5—Flavor macro HDL_EXPR

4.3.2.5 HDL_RANGE

Some HDLs provide special syntax for referring to the range of values that a variable or index may take on. Flavor macro HDL_RANGE captures this possibility, as shown in Box 5. Unlike other flavor macros, this one only includes options for those languages that support special range syntax.

```
Flavor Macro HDL_RANGE =
    VHDL: VHDL_Expression
```

Box 6—Flavor macro HDL_RANGE

NOTE—Flavor macro HDL_RANGE only applies in a VHDL context, because VHDL is the only language that includes special syntax for referring to previously defined ranges.

4.3.2.6 AND_OP, OR_OP, and NOT_OP

Each flavor of PSL overloads the underlying HDL's symbols for the logical (i.e., Boolean) conjunction, disjunction, and negation operators so the same operators are used for conjunction and disjunction of Boolean expressions and for conjunction, disjunction, and negation of properties. The definitions of `AND_OP`, `OR_OP`, and `NOT_OP` reflect this overloading, as shown in Box 7.

```
Flavor Macro AND_OP =
    SystemVerilog: && / Verilog: && / VHDL: and / GDL: &
Flavor Macro OR_OP =
    SystemVerilog: || / Verilog: || / VHDL: or / GDL: |
Flavor Macro NOT_OP =
    SystemVerilog: ! / Verilog: ! / VHDL: not / GDL: !
```

Box 7—Flavor macros AND_OP, OR_OP, and NOT_OP

4.3.2.7 RANGE_SYM, MIN_VAL, and MAX_VAL

Within properties it is possible to specify a range of integer values representing the number of cycles or number of repetitions that are allowed to occur. PSL adopts the general form of range specification from the underlying HDL, as reflected in the definition of `RANGE_SYM`, `MIN_VAL`, and `MAX_VAL` shown in Box 8.

```
Flavor Macro RANGE_SYM =
    SystemVerilog: : / Verilog: : / VHDL: to / GDL: ..
Flavor Macro MIN_VAL =
    SystemVerilog: 0 / Verilog: 0 / VHDL: 0 / GDL: null
Flavor Macro MAX_VAL =
    SystemVerilog: $ / Verilog: inf / VHDL: inf / GDL: null
```

Box 8— Flavor macros RANGE_SYM, MIN_VAL, and MAX_VAL

However, unlike HDLs, in which ranges are always finite, a range specification in PSL may have an infinite upper bound. For this reason, the definition of `MAX_VAL` includes the keyword **inf**, representing *infinite*.

4.3.2.8 LEFT_SYM and RIGHT_SYM

In replicated properties, it is possible to specify the replication index `Name` as a vector of boolean values. PSL allows this specification to take the form of an array reference in the underlying HDL, as reflected in the definition of `LEFT_SYM` and `RIGHT_SYM` shown in Box 9.

```
Flavor Macro LEFT_SYM =
    SystemVerilog: [  / Verilog: [  / VHDL: (  / GDL: (
Flavor Macro RIGHT_SYM =
    SystemVerilog: ]  / Verilog: ]  / VHDL: )  / GDL: )
```

Box 9—Flavor macro LEFT_SYM and RIGHT_SYM

4.3.2.9 DEF_SYM

Finally, as in the underlying HDL, PSL can declare new named objects. To make the syntax of such declarations consistent with those in the HDL, PSL adopts the symbol used for declarations in the underlying HDL, as reflected in the definition of `DEF_SYM` shown in Box 10.

```
Flavor Macro DEF_SYM  =
    SystemVerilog: = / Verilog: = / VHDL: is / GDL: :=
```

Box 10—Flavor macro DEF_SYM

4.4 Semantics

In this document, the following terms are used to describe the semantics of the language:

— *shall* indicates a required aspect of the PSL specification and *can* indicates an optional aspect of the PSL specification.
— In the informal (i.e., English) description of the semantics of the temporal layer, *holds* (or *doesn't hold*) indicates that the design does (or does not) behave in the manner specified by a property.

4.4.1 Clocked vs. unclocked evaluation

PSL properties can be modified by using a *clock expression* to indicate that time shall be measured in clock cycles of the clock expression. Such a property is a *clocked property.* The meaning of a clocked property is not affected by the granularity of time as seen by the verification tool. Thus, a clocked property shall give the same result for cycle-based and event-based verification.

Properties that are not modified by a clock expression are *unclocked properties*.

PSL does not dictate how time ticks for an unclocked property. Thus, unclocked properties are used to reason about the sequence of signal values as seen by the verification tool being used. For instance, a cycle-based simulator sees a sequence of signal values calculated cycle-by-cycle, while an event-based simulator running on the same design sees a more detailed sequence of signal values.

4.4.2 Safety vs. liveness properties

A *safety property* is a property that specifies an invariant over the states in a design. The invariant is not necessarily limited to a single cycle, but it is bounded in time. Loosely speaking, a safety property claims that "something bad" does not happen. More formally, a safety property is a property for which any path violating the property has a finite prefix such that every extension of the prefix violates the property. For example, the property "whenever signal `req` is asserted, signal `ack` is asserted within 3 cycles" is a safety property.

A *liveness property* is a property that specifies an eventuality that is unbounded in time. Loosely speaking, a liveness property claims that "something good" eventually happens. More formally, a liveness property is a property for which any finite path can be extended to a path satisfying the property. For example, the property "whenever signal `req` is asserted, signal `ack` is asserted sometime in the future" is a liveness property.

4.4.3 Linear vs. branching logic

PSL contains both properties that use linear semantics as well as those that use branching semantics. The former are properties of the PSL Foundation Language, while the latter belong to the Optional Branching Extension. Properties with *linear semantics* reason about computation paths in a design and can be checked in simulation, as well as in formal verification. Properties with *branching semantics* reason about computation trees and can be checked only in formal verification.

While the linear semantics of PSL are the ones most used in properties, the branching semantics add important expressive power. For instance, branching semantics are sometimes required to reason about deadlocks.

4.4.4 Simple subset

PSL can express properties that cannot be easily evaluated in simulation, although such properties can be addressed by formal verification methods.

In particular, PSL can express properties that involve branching or parallel behavior, which tend to be more difficult to evaluate in simulation, where time advances monotonically along a single path. The simple subset of PSL is a subset that conforms to the notion of monotonic advancement of time, left to right through the property, which in turn ensures that properties within the subset can be simulated easily. The simple subset of PSL contains any PSL FL property meeting all of the following conditions:

— The operand of a negation operator is a Boolean.
— The operand of a `never` operator is a Boolean or a Sequence.
— The operand of an `eventually!` operator is a Boolean or a Sequence.
— The left-hand side operand of a logical and operator is a Boolean.
— The left-hand side operand of a logical or operator is a Boolean.
— The left-hand side operand of a logical implication (`->`) operator is a Boolean.
— Both operands of a logical iff (`<->`) operator are Boolean.
— The right-hand side operand of a non-overlapping `until*` operator is a Boolean.
— Both operands of an overlapping `until*` operator are Boolean.
— Both operands of a `before*` operator are Boolean.

All other operators not mentioned above are supported in the simple subset without restriction. In particular, all of the `next_event` operators and all forms of suffix implication are supported in the simple subset.

4.4.5 Finite-length versus infinite-length behavior

The semantics of PSL allow us to decide whether a PSL property holds on a given behavior. How the outcome of this problem 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, PSL 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 any 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 any 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

4.4.6 The concept of strength

PSL uses the term *strong* in two different ways: an operator may be strong, and the satisfaction of an assertion on a path may be strong. While the two are related, the use of the concept of strength in each context is best understood first in isolation. Each is presented below, then the relation between them is explained.

4.4.6.1 Strong vs. weak operators

Some operators have a terminating condition that comes at an unknown time. For example, the property "busy shall be asserted until done is asserted" is expressed using an operator of the `until` family, which states that signal `busy` shall stay asserted until the signal `done` is asserted. The specific cycle in which signal `done` is asserted is not specified.

Operators such as these come in both strong and weak forms. The *strong form* requires that the terminating condition eventually occur, while the *weak form* makes no requirements about the terminating condition. For example, the strong and weak forms of "busy shall be asserted until done is asserted" are (`busy until! done`) and (`busy until done`), respectively. The former states that `busy` shall be asserted until `done` is asserted and that `done` shall eventually be asserted). The latter states that `busy` shall be asserted until `done` is asserted and that if done is never asserted, then `busy` shall stay asserted forever.

The distinction between weak and strong operators is related to the distinction between safety and liveness properties. A property that uses a non-negated strong operator is a liveness property, while one that contains only non-negated weak operators is a safety property.

4.4.6.2 Strong satisfaction

Strong satisfaction is related to the status of a property on a finite path, as seen for example in simulation. If a property holds on a finite path, and in addition, we know that the property will hold on any extension of the path, we say that the property is satisfied strongly. For instance, the property (expressed in English) *p is eventually asserted* holds strongly on a finite path on which p is asserted at some point. The property (expressed in English) *p is always asserted* does not hold strongly on such a path (and indeed holds strongly on no finite path), because we can never be sure that extending the path will not cause the property to fail.

4.4.6.3 Relating the two concepts of strength

The relationship between the strength of an operator and the strength of satisfaction of a property is as follows: assume we have a property `p` such that the only negation appears on boolean expressions. Replace all operators in `p` with their strong versions, and call the result `p_s`. Then property `p` holds strongly on a finite path iff property `p_s` holds on the path.

5. Boolean layer

The *Boolean layer* consists of expressions that represent design behavior. These expressions build upon the expression capabilities of the HDL(s) used to describe the design under consideration. All expressions in the Boolean layer evaluate immediately, at an instant in time.

Expressions may be of various HDL-specific data types. Certain classes of HDL data types are distinguished in PSL, due to their specific roles in describing behavior. Each class of data types in PSL corresponds to a set of specific data types in the underlying HDL design.

Expressions may involve HDL-specific expression syntax or PSL-defined operators and built-in functions. PSL-defined operators and built-in functions map onto underlying HDL-specific operations, as appropriate for the HDL context and the data type of the expression.

HDL-specific expressions are not redefined by PSL. Rather, PSL uses a subset of the existing IEEE standards. See Chapter 8 (Modeling layer) for details.

5.1 Expression Type Classes

Five classes of expression are distinguished in PSL: Bit, Boolean, BitVector, Numeric, and String expressions. Each of these correspond to a set of specific data types in the underlying HDL context, and an interpretation of the values of those data types. Some PSL expressions and built-in functions require operands that belong to specific expression classes. Others take operands of any type.

```
Any_Type ::=
  HDL_or_PSL_Expression
```

Box 11—Any type expression

5.1.1 Bit expressions

Bit expressions represent the values of individual signals or memory elements in the design. The data types used in bit expressions include types that model bits as strictly binary (having values in {0,1}) as well as multi-valued logic types, with values in {X, 0, 1, Z}.

```
Bit ::=
  bit_HDL_or_PSL_Expression
```

Box 12—Bit expression

In Verilog, the built-in logic type is a Bit type.

In SystemVerilog, the built-in types *bit* and *logic* are Bit types.

In VHDL, type *STD.Standard.Bit*, and type *IEEE.Std_Logic_1164.std_ulogic*, as well as subtypes thereof, are Bit types.

5.1.2 Boolean expressions

Boolean expressions, for which the Boolean layer is named, describe states of the design, in terms of signals, values, and their relationships. They represent simple properties, which can be composed using temporal operators to create temporal properties.

```
Boolean ::=
  boolean_HDL_or_PSL_Expression
```

Box 13—Boolean expression

Boolean expressions may be dynamic, i.e., they may contain signals whose values change over time. Boolean expressions may have subexpressions of any type.

In VHDL, type STD.Standard.Boolean is a Boolean type.

Any Bit type is interpretable as a Boolean type. For Verilog and SystemVerilog, a BitVector expression may also appear where a Boolean expression is required, in which case the expression is interpreted as True or False according to the rules of Verilog and SystemVerilog, respectively, for interpreting an expression that appears as the condition of an if statement.

The return value from a PSL expression or built-in function that returns a Boolean value is of the appropriate type for the context. For Verilog, the return value is of the built-in logic type; for SystemVerilog, the return value is of the built-in type logic, for VHDL, the return value is of type *STD.Standard.Boolean*.

Literals True and False represent the corresponding literals in the underlying HDL Boolean type (or Bit type interpreted as a Boolean type) involved in a given expression.

A Boolean expression is required wherever the nonterminal Boolean appears in the syntax.

5.1.3 BitVector expressions

BitVector expressions represent words composed of bits, of various widths.

```
BitVector ::=
  bitvector_HDL_or_PSL_Expression
```

Box 14—BitVector expression

In Verilog, and in SystemVerilog, any reg, wire, or net type, and any word in a memory, is interpretable as a BitVector type.

In VHDL, any type that is a one-dimensional array of a Bit type is interpretable as a BitVector type.

The return value from a PSL built-in function that returns a BitVector value is of the appropriate type for the context. For Verilog, the return value is a vector of the built-in logic type; for SystemVerilog, the return value is a vector of the built-in type logic; for VHDL, the return value is of type IEEE.Std_Logic_1164.std_ulogic_vector.

5.1.4 Numeric expressions

Numeric expressions represent integer constants such as cycle or occurrence counts that are part of the definition of a temporal property.

```
Number ::=
  numeric_HDL_or_PSL_Expression
```

Box 15—Numeric expression

In Verilog, any BitVector expression that contains no unknown bit values is interpretable as a Numeric expression. In SystemVerilog, any integral type is interpretable as a Numeric type. In VHDL, any expression of an integer type is interpretable as a Numeric expression.

The return value from a PSL built-in function that returns a Numeric value is of the appropriate type for the context. For Verilog, the return value is a vector of the built-in logic type; for SystemVerilog, the return value is of the built-in type int; for VHDL, the return value is of type STD.Standard.Integer.

A Numeric expression is required wherever the nonterminal Number appears in the syntax.

Restrictions

Numeric expressions must be statically evaluable--signals or variables that can change value over time are not allowed in expressions that must be Numeric. Numeric expressions are always non-negative; in some cases they must be non-zero as well.

5.1.5 String expressions

String expressions represent text messages that are attached to a PSL directive to help in debugging.

```
String ::=
  string_HDL_or_PSL_Expression
```

Box 16—String expression

In Verilog, any string literal is a String expression. In SystemVerilog, any expression of type string is a String expression. In VHDL, any expression of type *STD.Standard.String* is a String expression.

A String expression is required wherever the nonterminal String appears in the syntax.

5.2 Expression forms

Expressions in the Boolean Layer are built from HDL expressions, PSL expressions, PSL built-in functions, endpoint instances, and union expressions, as Box 16 illustrates.

```
HDL_or_PSL_Expression ::=
    HDL_Expression
  | PSL_Expression
  | Built_In_Function_Call
  | Union_Expression
  | Endpoint_Instance
```

Box 17—HDL or PSL Expression

In each flavor of PSL, at any place where an HDL subexpression may appear within an HDL or PSL expression, the grammar of the corresponding HDL is extended to allow any form of HDL or PSL expression. Thus HDL expressions, PSL expressions, built-in functions, endpoints, and union expressions may all be used as subexpressions within HDL or PSL expressions.

NOTE—Subexpressions of a Boolean expression may be of any type supported by the corresponding HDL.

5.2.1 HDL expressions

An HDL expression may be used wherever a Bit, Boolean, BitVector, Numeric, or String expression is required, provided that the type of the expression is (or is interpretable as) the required type.

```
HDL_Expression  ::=
  HDL_EXPR

Flavor Macro HDL_EXPR =
  SystemVerilog: SystemVerilog_Expression
 / Verilog: Verilog_Expression
 / VHDL: VHDL_Expression
 / GDL: GDL_Expression
```

Box 18—HDL expression

Informal semantics

The meaning of an HDL expression in a PSL context is determined by the meanings of the names and operator symbols of the HDL expression.

For each name in the HDL expression, the meaning of the name is determined as follows.

a) If the current verification unit contains a (single) declaration of this name, then the object created by that declaration is the meaning of this name.
b) Otherwise, if the transitive closure with respect to inheritance of all verification units inherited by the current verification unit contains a (single) declaration of this name, then the object created by that declaration is the meaning of this name.
c) Otherwise, if the default verification mode contains a (single) declaration of this name, then the object created by that declaration is the meaning of this name.
d) Otherwise, if this name has an unambiguous meaning at the end of the design module or instance associated with the current verification unit, then that meaning is the meaning of this name.
e) Otherwise, this name has no meaning.

It is an error if more than one declaration of a given name appears in the current verification unit (in step (a)), or in the transitive closure of all inherited verification units (in step (b)), or in the default verification mode (in step (c)), or if the name is ambiguous at the end of the associated design module or instance (in step (d)).

For each operator symbol in the HDL expression, the meaning of the operator symbol is determined as follows.

— For the SystemVerilog, Verilog, and GDL flavors, this operator symbol has the same meaning as the corresponding operator symbol in the HDL.
— For the VHDL flavor, if this operator symbol has an unambiguous meaning at the end of the design unit or component instance associated with the current verification unit, then that meaning is the meaning of this operator symbol.
—
— Otherwise, this operator symbol has no meaning.

See 7.2 for an explanation of verification units and modes.

5.2.2 PSL expressions

PSL defines a collection of operators that represent underlying HDL operators..

```
HDL_or_PSL_Expression ::=
   PSL_Expression

PSL_Expression ::=
   Boolean -> Boolean
 | Boolean <-> Boolean
```

Box 19—PSL expression

Both PSL expression operators involve operands that are (or are interpretable as) Boolean. Each produces a Boolean result.

Informal semantics

Each of these operators represent, or map to, equivalent operators defined by the HDL in which the relevant portion of the design is described, as appropriate for the data types of the operands.

In a Verilog or SystemVerilog context, the mapping is as follows. PSL expression `a -> b` maps to the equivalent expression `(!a || b),` and PSL expression `a <-> b` maps to the equivalent expression
`((a && b) || (!a && !b)).`

In a VHDL context, the mapping is as follows. PSL expression `a -> b` maps to the equivalent expression
`(not a or b)`, and PSL expression `a <-> b` maps to the equivalent expression
`((a and b) or (not a and not b)).`

In the GDL flavor, these operators are native operators, so no mapping is involved.

5.2.3 Built-in functions

PSL defines a collection of built-in functions that detect typically interesting conditions.

```
Built_In_Function_Call ::=
    prev (Any_Type [ , Number ] )
  | next ( Any_Type )
  | stable ( Any_Type )
  | rose ( Bit )
  | fell ( Bit )
  | isunknown ( BitVector )
  | countones ( BitVector )
  | onehot ( BitVector )
  | onehot0 ( BitVector )
```

Box 20—Built-in functions

There are two classes of built-in functions. Functions `prev()`, `next()`, `stable()`, `rose()`, and `fell()` all have to do with the values of expressions over time. Functions `isunknown()`, `countones()`, `onehot()`, and `onehot0()` all have to do with the values of bits in a vector at a given instant.

5.2.3.1 prev()

The built-in function `prev()` takes an expression of any type as argument and returns a previous value of that expression. With a single argument, the built-in function `prev()` gives the value of the expression in the previous cycle, with respect to the clock of its context. If a second argument is specified and has the value `i`, the built-in function `prev()` gives the value of the expression in the i^{th} previous cycle, with respect to the clock of its context.

The clock context may be provided by the PSL property in which the function call is nested, or by a relevant default clock declaration. If the context does not specify a clock, the relevant clock is that corresponding to the granularity of time as seen by the verification tool.

NOTE—The first argument of `prev()` is not necessarily a Boolean expression. For example, if the argument to `prev()` is a bit vector, then the result is the previous value of the entire bit vector.

Restrictions

If a call to `prev()` includes a Number, it must be a positive Number that is statically evaluatable.

Example

In the timing diagram below, the function call `prev(a)` returns the value 1 at times 3, 4, and 6, and the value 0 at other times, if it does not have a clock context. In the context of clock `clk`, the `call prev(a)` returns the value 1 at times 5 and 7, and the value 0 at other tick points. In the context of clock `clk`, the call `prev(a,2)` returns the value 1 at time 7, and 0 at other tick points.

```
time    0 1 2 3 4 5 6 7
-----------------------
clk     0 1 0 1 0 1 0 1
a       0 0 1 1 0 1 0 0
```

5.2.3.2 next()

The built-in function `next()` gives the value of a signal of any type at the next cycle, with respect to the finest granularity of time as seen by the verification tool. In contrast to the built-in functions `prev()`, `stable()`, `rose()`, and `fell()`, the function `next()` is not affected by the clock of its context.

Restrictions

The argument of `next()` shall be the name of a signal; an expression other than a simple name is not allowed. A call to `next()` can only be used on the right-hand-side of an assignment to a memory element (register or latch). It cannot be used on the right-hand-side of an assignment to a combinational signal, nor can it be used directly in a property.

Example

In the timing diagram below, the function call `next(a)` returns the value 1 at times 1, 2, and 4.

```
time    0 1 2 3 4 5 6 7
-----------------------
clk     0 1 0 1 0 1 0 1
a       0 0 1 1 0 1 0 0
```

5.2.3.3 stable()

The built-in function `stable()` takes an expression of any type as argument and produces a Boolean result that is true if the argument's value is the same as it was at the previous cycle, with respect to the clock of its context.

The clock context may be provided by the PSL property in which the function call is nested, or by a relevant default clock declaration. If the context does not specify a clock, the relevant clock is that corresponding to the granularity of time as seen by the verification tool.

The function `stable()` can be expressed in terms of the built-in function `prev()` as follows: `stable(e)` is equivalent to the Verilog or SystemVerilog expression (`prev(e) == e`), and is equivalent to the VHDL expression (`prev(e) = e`), where `e` is any expression. The function `stable()` can be used anywhere a Boolean is required.

Example

In the timing diagram below, the function call `stable(a)` is true at times 1, 3, and 7, and at no other time if it does not have a clock context. In the context of clock `clk`, the function call `stable(a)` is true at the tick of `clk` at time 5 and at no other tick point of `clk`.

```
time    0 1 2 3 4 5 6 7
-----------------------
clk     0 1 0 1 0 1 0 1
a       0 0 1 1 0 1 0 0
```

5.2.3.4 rose()

The built-in function `rose()` takes a Bit expression as argument and produces a Boolean result that is true if the argument's value is 1 at the current cycle and 0 at the previous cycle, with respect to the clock of its context, otherwise it is false.

The clock context may be provided by the PSL property in which the function call is nested, or by a relevant default clock declaration. If the context does not specify a clock, the relevant clock is that corresponding to the granularity of time as seen by the verification tool.

The function `rose()` can be expressed in terms of the built-in function `prev()` as follows: `rose(b)` is equivalent to the Verilog or SystemVerilog expression (`prev(b)==1'b0 && b==1'b1`), and is equivalent to the VHDL expression (`prev(b)='0' and b='1'`), where `b` is a Bit signal. The function `rose(b)` can be used anywhere a Boolean is required.

NOTE—For a given property f and signal clk, f@rose(clk), f@(posedge clk), and f@(rising_edge(clk)) all have equivalent semantics, provided that signal clk takes on only 0 and 1 values, and no signal in f changes at the same time as clk (i.e., there are no race conditions).

If signal clk can take on X or Z values, then the semantics of f@(posedge clk) may differ from those of f@rose(clk) and f@(rising_edge(clk)), because (posedge clk) will generate an event on 0->X, X->1, 0->Z, and Z->1 transitions of clk, whereas rose(clk) and rising_edge(clk) will ignore these transitions.

If at least one signal appearing in f changes at the same time as clk, then the semantics of f@(posedge clk), f@rose(clk), and f@(rising_edge(clk)) may be different, due to differences in their respective handling of race conditions.

Example

In the timing diagram below, the function call `rose(a)` is true at times 2 and 5 and at no other time, if it has no clock context. In the context of clock `clk`, the function call `rose(a)` is true at the tick of `clk` at time 3 and at no other tick point of `clk`.

```
time    0 1 2 3 4 5 6 7
-----------------------
clk     0 1 0 1 0 1 0 1
a       0 0 1 1 0 1 0 0
```

5.2.3.5 fell()

The built-in function `fell()` takes a Bit expression as argument and produces a Boolean result that is true if the argument's value is 0 at the current cycle and 1 at the previous cycle, with respect to the clock of its context, otherwise it is false.

The clock context may be provided by the PSL property in which the function call is nested, or by a relevant default clock declaration. If the context does not specify a clock, the relevant clock is that corresponding to the granularity of time as seen by the verification tool.

The function `fell()` can be expressed in terms of the built-in function `prev()` as follows: `fell(b)` is equivalent to the Verilog or SystemVerilog expression (`prev(b)==1'b1 && b==1'b0`), and is equivalent to the VHDL expression (`prev(b)='1' and b='0'`), where `b` is a Bit signal. The function `fell(b)` can be used anywhere a Boolean is required.

NOTE—For a given property f and signal clk, f@fell(clk), f@(negedge clk), and f@(falling_edge(clk)) all have equivalent semantics, provided that signal clk takes on only 0 and 1 values, and no signal in f changes at the same time as clk (i.e., there are no race conditions).

If signal clk can take on X or Z values, then the semantics of f@(negedge clk) may differ from those of f@fell(clk) and f@(falling_edge(clk)), because (negedge clk) will generate an event on 1->X, X->0, 1->Z, and Z->0 transitions of clk, whereas fell(clk) and falling_edge(clk) will ignore these transitions.

If at least one signal appearing in f changes at the same time as clk, then the semantics of f@(negedge clk), f@fell(clk), and f@(falling_edge(clk)) may be different, due to differences in their respective handling of race conditions.

Example

In the timing diagram below, the function call fell(a) is true at times 4 and 6 and at no other time if it does not have a clock context. In the context of clock clk, the function call fell(a) is true at the tick of clk at time 7 and at no other tick point of clk.

```
time    0 1 2 3 4 5 6 7
-----------------------
clk     0 1 0 1 0 1 0 1
a       0 0 1 1 0 1 0 0
```

5.2.3.6 isunknown()

The built-in function `isunknown()` takes a BitVector as argument. It returns True if the argument contains any bits that have “unknown” values (i.e., values other than 0 or 1); otherwise it returns False.

Function `isunknown()` can be used anywhere a Boolean is required.

5.2.3.7 countones()

The built-in function `countones()` takes a BitVector as argument. It returns a count of the number of bits in the argument that have the value 1.

Bits that have unknown values are ignored.

NOTE—Although function `countones()` returns a Numeric result, it can only be used where a Number is required if it has a statically evaluable argument.

5.2.3.8 onehot(), onehot0()

The built-in function `onehot()` takes a BitVector as argument. It returns True if the argument contains exactly one bit with the value 1; otherwise it returns False.

The built-in function `onehot0()` takes a BitVector as argument. It returns True if the argument contains at most one bit with the value 1; otherwise it returns False.

For either function, bits that have unknown values are ignored.

Functions `onehot()` and `onehot0()` can be used anywhere a Boolean is required

5.2.4 Union expressions

The union operator specifies two values, shown in Box 20, either of which can be the value of the resulting expression.

```
Union_Expression ::=
  Any_Type union Any_Type
```

Box 21—Union expression

Restrictions:

The two operands must be of the same underlying HDL type.

Example

```
a = b union c;
```

This is a non-deterministic assignment of either `b` or `c` to variable or signal `a`.

5.2.4.1 Endpoints

PSL defines a special variable called an endpoint, which signals the completion of a sequence. Endpoint declarations and instantiations are described in 6.1.4.1 and 6.1.4.2, respectively.

5.3 Clock expressions

Booleans (either Boolean HDL expressions, or PSL expressions) can be used as clock expressions, which indicate when other Boolean expressions are evaluated.

```
Clock_Expression :=
      boolean_Name
    | boolean_Built_In_Function_Call
    | Endpoint_Instance
    | ( Boolean )
    | ( HDL_CLK_EXPR )

Flavor Macro HDL_CLK_EXPR =
      SystemVerilog: SystemVerilog_Event_Expression
    / Verilog: Verilog_Event_Expression
    / VHDL: VHDL_Expression
    / GDL: GDL_Expression
```

Box 22—Clock expression

Any PSL expression that is a Boolean expression can be enclosed in parentheses and used as a clock expression. In particular, PSL built-in functions `rose()` and `fell()`, and endpoint instances, can be used as clock expressions. Boolean names, built-in function calls, and endpoint instances can also be used as clock expressions without enclosing them in parentheses.

In the SystemVerilog flavor, any expression that SystemVerilog allows to be used as the condition in an if statement can be used as a clock expression. In addition, any SystemVerilog *event expression* that is not a single

Boolean expression can be used as a clock expression. Such a clock expression is considered to hold in a given cycle *iff* it generates an event in that cycle.

In the Verilog flavor, any expression that Verilog allows to be used as the condition in an `if` statement can be used as a clock expression. In addition, any Verilog event expression that is not a single Boolean expression can be used as a clock expression. Such a clock expression is considered to hold in a given cycle *iff* it generates an event in that cycle.

In the VHDL flavor, any expression that VHDL allows to be used as the condition in an `if` statement can be used as a clock expression.

In the GDL flavor, any expression that GDL allows to be used as the condition in an `if` statement can be used as a clock expression.

5.4 Default clock declaration

A *default clock declaration*, shown in Box 23, specifies a clock expression for directives that have an outermost property or sequence that has no explicit clock expression.

```
PSL_Declaration ::=
    Clock_Declaration
Clock_Declaration ::=
    default clock DEF_SYM Clock_Expression ;
```

Box 23—Default clock declaration

Restrictions

At most one default clock declaration shall appear in a given verification unit.

Informal semantics

If the outermost property of an `assert`, `assume`, or `assume_guarantee` directive has no explicit clock expression, then the clock expression for that property is given by the applicable default clock declaration, if one exists; otherwise the clock expression for the property is the expression `True`.

Similarly, if the outermost sequence of a `cover, restrict`, or `restrict_guarantee` directive has no explicit clock expression, then the clock expression for that sequence is determined by the applicable default clock declaration, if one exists; otherwise the clock expression for the sequence is the expression `True`.

The applicable default clock declaration is determined as follows.

a) If the current verification unit contains a (single) default clock declaration, then that is the applicable default clock declaration.
b) Otherwise, if the transitive closure with respect to inheritance of all verification units inherited by the current verification unit contains a (single) default clock declaration, then that is the applicable default clock declaration.
c) Otherwise, if the default verification mode contains a (single) default clock declaration, then that is the applicable default clock declaration.
d) Otherwise, no applicable default clock declaration exists.

It is an error if, in step a), more than one default clock declaration appears in the current verification unit; or if, in step b), more than one default clock declaration appears in the transitive closure of all inherited verification units; or if, in step c), more than one default clock declaration appears in the default verification mode.

Example

```
   default clock = (posedge clk1);

   assert always (req -> next ack);
   cover {req; ack; !req; !ack};
```

is equivalent to

```
   assert (always (req -> next ack))@(posedge clk);
   cover {req; ack; !req; !ack} @(posedge clk1);
```

NOTE—A property f@True, in the context of a default clock, has the same effect as property f, without a default clock. The clock expression True effectively masks the default clock so that it has no effect on property f.

NOTE—The default clock declaration

```
     default clock = True ;
```

has the same effect as having no default clock declaration.

6. Temporal layer

The temporal layer is used to define properties, which describe behavior over time. Properties can describe the behavior of the design or the behavior of the external environment.

A property is built from four types of building blocks:

- — Boolean expressions
- — clock expressions
- — sequential expressions (which are themselves built from Boolean expressions)
- — subordinate properties

Boolean expressions and clock expressions are part of the Boolean layer; they are described in section 5. Sequential expressions are described in 6.1 and properties in 6.2.

Some terms used in this section and their definitions are:

holds tightly: The term used to talk about the meaning of a sequential expression (SERE). Sequential expressions are evaluated over finite paths (behaviors). The definition of holds tightly captures the meaning of a SERE by determining the finite paths that "match" the SERE. The meaning of a SERE depends on the operators and sub-SEREs that constitute the SERE. Thus, the definition of holds tightly is given in the sub-sections of Section 6.1; for each SERE operator, the sub-section describing that operator defines the finite paths on which a SERE that combines other SEREs using that operator holds tightly, given the meaning of these subordinate SEREs. Formally, a sequential expression holds tightly on a given trace iff that trace tightly models the sequential expression, as defined in Appendix B.

For example, {a;b;c} holds tightly on a path iff the path is of length three, where 'a' is true in the first cycle, 'b' is true in the second and 'c' is true in the third. The SERE {a[*];b} holds tightly on a path iff 'b' is true in the last cycle of the path, and 'a' is true in all preceding cycles.

holds: The term used to talk about the meaning of a Boolean expression, sequential expression, or property. A Boolean expression, sequential expression, or property is evaluated over the first cycle of a finite or infinite path. The definition of holds captures the meaning of a Boolean expression, sequential expressions or property by determining the paths (starting at the current cycle) that "obey" them. The meaning of a property depends on the operators and subordinate properties that constitute the property. Thus, the definition of holds is given in the sub-sections of Section 6.2; for each operator it is defined, in the sub-section describing that operator, which are the paths the composed property holds on (at their first state). Formally, a Boolean expression, sequential expression, or property holds on a given trace iff the trace models (or satisfies) that Boolean expression, sequential expression, or property, as defined in Appendix B.

For example, a Boolean expression 'p' holds in the first cycle of a path iff 'p' evaluates to *True* in the first cycle. A SERE holds on the first cycle of a path iff it holds tightly on a prefix of that path. The sequential expression {a;b;c} holds on a first cycle of a path iff 'a' holds on the first cycle, 'b' holds on the second cycle and 'c' holds on the third. Note that the path itself may be of length more than 3. The sequential expression {a[*];b} holds in the first cycle of a path iff: 1) the path contains a cycle in which 'b' holds, and 2) 'a' holds in all cycles before that cycle. It is not necessary that the cycle in which 'b' holds is the last cycle of the path (contrary to the requirement for {a[*];b} to hold tightly on a path). Finally, the property 'always p' holds in a first cycle of a path iff 'p' holds in that cycle and in every subsequent cycle.

describes: A Boolean expression, sequential expression, or property describes the set of behavior for which the Boolean expression, sequential expression, or property holds.

occurs: A Boolean expression is said to "occur" in a cycle if it holds in that cycle. For example, "the next occurrence of the Boolean expression" refers to the next cycle in which the Boolean expression holds.

starts: A sequential expression starts at the first cycle of any behavior for which it holds. In addition, a sequential expression starts at the first cycle of any behavior that is the prefix of a behavior for which it holds. For example, if `a` holds at cycle 7 and `b` holds at every cycle from 8 onward, then the sequential expression `{a;b[*];c}` starts at cycle 7.

completes: A sequential expression completes at the last cycle of any design behavior on which it holds tightly. For example, if `a` holds at cycle 3, `b` holds at cycle 4, and `c` holds at cycle 5, then the sequence `{a;b;c}` completes at cycle 5. Similarly, given the behavior `{a;b;c}`, the property `a before c` completes when `c` occurs.

NOTE—A sequence that holds eventually completes, while a sequence that starts may or may not complete.

terminating condition: A Boolean expression, the occurrence of which causes a property to complete.

terminating property: A property that, when it holds, causes another property to complete.

NOTE—These terms are used to describe the semantics of the temporal layer as precisely as possible in English. In any case where the English description is ambiguous or seems to contradict the formal semantics provided in Appendix B, the formal semantics take precedence.

6.1 Sequential expressions

6.1.1 Sequential Extended Regular Expressions (SEREs)

Sequential Extended Regular Expressions (*SERE*s), shown in Box 24, describe single- or multi-cycle behavior built from a series of Boolean expressions.

```
SERE ::=
        Boolean
      | Sequence
      | Sequence_Instance
```

Box 24—SEREs and Sequences

The most basic SERE is a Boolean expression. A Sequence (see 6.1.2) and a Sequence Instance (see 6.1.3.2) are also SEREs.

More complex sequential expressions are built from Boolean expressions using various SERE operators. These operators are described in the subsections that follow.

NOTE—SEREs are grouped using curly braces (`{}`), as opposed to Boolean expressions that are grouped using parentheses (`()`). See section 6.1.2.4.

6.1.1.1 Simple SEREs

Simple SEREs represent a single thread of subordinate behaviors, occurring in successive cycles.

6.1.1.1.1 SERE concatenation (;)

The *SERE concatenation* operator (;), shown in Box 25, constructs a SERE that is the concatenation of two other SEREs.

```
SERE ::=
    SERE ; SERE
```

Box 25—SERE concatenation operator

The right operand is a SERE that is concatenated after the left operand, which is also a SERE.

Restrictions

None.

Informal semantics

For SEREs A and B:

> A;B holds tightly on a path iff there is a future cycle *n*, such that A holds tightly on the path up to and including the *n*th cycle and B holds tightly on the path starting at the *n*+1th cycle.

6.1.1.1.2 SERE fusion (:)

The *SERE fusion* operator (:), shown in Box 26, constructs a SERE in which two SEREs overlap by one cycle. That is, the second starts at the cycle in which the first completes.

```
SERE ::=
    SERE : SERE
```

Box 26—SERE fusion operator

The operands of : are both SEREs.

Restrictions

None.

Informal semantics

For SEREs A and B:

> A:B holds tightly on a path iff there is a future cycle *n*, such that A holds tightly on the path up to and including the *n*th cycle and B holds tightly on the path starting at the *n*th cycle.

6.1.1.2 Compound SEREs

Compound SEREs represent a set of one or more threads of subordinate behaviors, starting from the same cycle, and occurring in parallel.

```
SERE ::=
        Compound_SERE

Compound_SERE ::=
        Repeated_SERE
      | Braced_SERE
      | Clocked_SERE
```

Box 27—Compound SEREs

A Repeated SERE, a Braced SERE, and a Clocked SERE (all of which are forms of Sequence; see 6.1.2) are Compound SEREs. Compound SERE operators allow the construction of additional forms of Compound SERE.

6.1.1.2.1 SERE or (|)

The *SERE or* operator (|), shown in Box 28, constructs a Compound SERE in which one of two alternative Compound SEREs hold at the current cycle.

```
Compound_SERE ::=
        Compound_SERE | Compound_SERE
```

Box 28—SERE or operator

The operands of | are both Compound SEREs.

Restrictions

None.

Informal semantics

For Compound SEREs A and B:

A | B holds tightly on a path iff at least one of A or B holds tightly on the path.

6.1.1.2.2 SERE non-length-matching and (&)

The *SERE non-length-matching and* operator (&), shown in Box 29, constructs a Compound SERE in which two Compound SEREs both hold at the current cycle, regardless of whether they complete in the same cycle or in different cycles.

```
Compound_SERE ::=
        Compound_SERE & Compound_SERE
```

Box 29—SERE non-length-matching and operator

The operands of & are both Compound SEREs.

Restrictions

None.

Informal semantics

For Compound SEREs A and B:

> A&B holds tightly on a path iff either A holds tightly on the path and B holds tightly on a prefix of the path or B holds tightly on the path and A holds tightly on a prefix of the path.

6.1.1.2.3 SERE length-matching and (&&)

The *SERE length-matching and* operator (&&), shown in Box 30, constructs a Compound SERE in which two Compound SEREs both hold at the current cycle, and furthermore both complete in the same cycle.

```
Compound_SERE ::=
        Compound_SERE && Compound_SERE
```

Box 30—SERE length-matching and operator

The operands of && are both Compound_SEREs.

Restrictions

None.

Informal semantics

For Compound_SEREs A and B:

> A&&B holds tightly on a path iff A and B both hold tightly on the path.

6.1.1.2.4 SERE within

The SERE within operator (`within`), shown in Box 31, constructs a Compound SERE in which the second Compound SERE holds at the current cycle, and the first Compound SERE starts at or after the cycle in which the second starts, and completes at or before the cycle in which the second completes.

```
Compound_SERE ::=
        Compound_SERE within Compound_SERE
```

Box 31—SERE within operator

The operands of `within` are both Compound SEREs.

Restrictions

None.

Informal semantics

For Compound SEREs A and B:

A `within` B holds tightly on a path iff the SERE {[*];A;[*]} && {B} holds tightly on the path.

6.1.2 Sequences

A sequence is a SERE that can appear at the top level of a declaration, directive, or property.

```
Sequence ::=
        Sequence_Instance
      | Repeated_SERE
      | Braced_SERE
      | Clocked_SERE
```

Box 32—Sequences

Sequence Instances are described in section 6.1.3.2. The remaining forms of Sequence are described in the following subsections.

6.1.2.1 SERE consecutive repetition ([*])

The *SERE consecutive repetition* operator (**[*]**), shown in Box 33, constructs repeated consecutive concatenation of a given Boolean or Sequence.

```
Repeated_SERE::=
        Boolean [* [ Count ] ]
      | Sequence [* [ Count ] ]
      | [* [ Count ] ]
      | Boolean [+]
      | Sequence [+]
      | [+]
Count ::=
      Number
      | Range
Range ::=
      Low_Bound RANGE_SYM High_Bound
Low_Bound ::=
      Number
      | MIN_VAL
High_Bound ::=
      Number
      | MAX_VAL
```

Box 33—SERE consecutive repetition operator

The first operand is a Boolean or Sequence to be repeated. The second operand gives the Count (a number or range) of repetitions.

If the Count is a number, then the repeated SERE describes exactly that number of repetitions of the first operand.

Otherwise, if the Count is a range, then the repeated SERE describes any number of repetitions of the first operand such that the number falls within the specified range. If the high value of the range (High_Bound) is specified as MAX_VAL, the repeated SERE describes at least as many repetitions as the low value of the range. If the low value of the range (Low_Bound) is specified as MIN_VAL, the repeated SERE describes at most as many repetitions as the high value of the range. If no range is specified, the repeated SERE describes any number of repetitions, including zero, i.e., the empty path is also described.

When there is no Boolean or Sequence operand and only a Count, the repeated SERE describes any path whose length is described by the second operand as above.

The notation [+] is a shortcut for a repetition of one or more times.

Restrictions

If the repeated SERE contains a Count, and the Count is a Number, then the Number shall be statically computable. If the repeated SERE contains a Count, and the Count is a Range, then each bound of the Range shall be statically computable, and the low bound of the Range shall be less than or equal to the high bound of the Range.

Informal semantics

For Boolean or Sequence `A` and numbers `n` and `m`:

— `A[*n]`holds tightly on a path iff the path can be partitioned into n parts, where A holds tightly on each part.
— `A[*n:m]`holds tightly on a path iff the path can be partitioned into between n and m parts, inclusive, where A holds tightly on each part.
— `A[*0:m]`holds tightly on a path iff the path is empty or the path can be partitioned into m or less parts, where A holds tightly on each part.
— `A[*n:inf]`holds tightly on a path iff the path can be partitioned into at least n parts, where A holds tightly on each part.
— `A[*0:inf]`holds tightly on a path iff the path is empty or the path can be partitioned into some number of parts, where A holds tightly on each part.
— `A[*]`holds tightly on a path iff the path is empty or the path can be partitioned into some number of parts, where A holds tightly on each part.
— `A[+]`holds tightly on a path iff the path can be partitioned into some number of parts, where A holds tightly on each part.
— `[*n]`holds tightly on a path iff the path is of length n.
— `[*n:m]`holds tightly on a path iff the length of the path is between n and m, inclusive.
— `[*0:m]`holds tightly on a path iff it is the empty path or the length of the path is m or less.
— `[*n:inf]`holds tightly on a path iff the length of the path is at least n.
— `[*0:inf]`holds tightly on any path (including the empty path).
— `[*]`holds tightly on any path (including the empty path).
— `[+]`holds tightly on any path of length at least one.

NOTE—If a repeated SERE begins with a Sequence that is itself a repeated SERE (e.g., a[*2][*3], where the repetition operator [*3] applies to the Sequence that is itself the repeated SERE a[*2]), the semantics are the same as if that Sequence were braced (e.g., {a[*2]}[*3]).

6.1.2.2 SERE non-consecutive repetition ([=])

The *SERE non-consecutive repetition* operator (**[=]**), shown in Box 34, constructs repeated (possibly non-consecutive) concatenation of a Boolean expression.

```
Repeated_SERE  ::=
        Boolean [= Count ]
Count  ::=
          Number
        | Range
Range ::=
        Low_Bound RANGE_SYM High_Bound
Low_Bound  ::=
        Number | MIN_VAL
High_Bound  ::=
        Number | MAX_VAL
```

Box 34—SERE non-consecutive repetition operator

The first operand is a Boolean expression to be repeated. The second operand gives the Count (a number or range) of repetitions.

If the Count is a number, then the repeated SERE describes exactly that number of repetitions.

Otherwise, if the Count is a range, then the repeated SERE describes any number of repetitions such that the number falls within the specified range. If the high value of the range (High_Bound) is specified as MAX_VAL, the repeated SERE describes at least as many repetitions as the low value of the range. If the low value of the range (Low_Bound) is specified as MIN_VAL, the repeated SERE describes at most as many repetitions as the high value of the range. If no range is specified, the repeated SERE describes any number of repetitions, including zero, i.e., the empty path is also described.

Restrictions

If the repeated SERE contains a Count, and the Count is a Number, then the Number shall be statically computable.

If the repeated SERE contains a Count, and the Count is a Range, then each bound of the Range shall be statically computable, and the low bound of the Range shall be less than or equal to the high bound of the Range.

Informal semantics

For Boolean `A` and numbers `n` and `m`:

— `A[=n]` holds tightly on a path iff A occurs exactly n times along the path.
— `A[=n:m]` holds tightly on a path iff A occurs between n and m times, inclusive, along the path.
— `A[=0:m]` holds tightly on a path iff A occurs m times or less along the path.
— `A[=n:inf]` holds tightly on a path iff A occurs at least n times along the path.
— `A[=0:inf]` holds tightly on a path iff A occurs any number of times along the path, i.e., A[=0:inf] holds tightly on any path.

NOTE—If a repeated SERE begins with a Sequence that is itself a repeated SERE (e.g., a[=2][*3], where the repetition operator [*3] applies to the Sequence that is itself the repeated SERE a[=2]), the semantics are the same as if that Sequence were braced (e.g., {a[=2]}[*3]).

6.1.2.3 SERE goto repetition ([->])

The *SERE goto repetition* operator (**[->]**), shown in Box 35, constructs repeated (possibly non-consecutive) concatenation of a Boolean expression, such that the Boolean expression holds on the last cycle of the path.

```
Repeated_SERE ::=
        Boolean [-> [ positive_Count ] ]
Count ::=
         Number
        | Range
Range ::=
        Low_Bound RANGE_SYM High_Bound
Low_Bound ::=
        Number | MIN_VAL
High_Bound ::=
        Number | MAX_VAL
```

Box 35—SERE goto repetition operator

The first operand is a Boolean expression to be repeated. The second operand gives the `Count` of repetitions.

If the Count is a number, then the repeated SERE describes exactly that number of repetitions.

Otherwise, if the Count is a range, then the repeated SERE describes any number of repetitions such that the number falls within the specified range. If the high value of the range (High_Bound) is specified as MAX_VAL, the repeated SERE describes at least as many repetitions as the low value of the range. If the low value of the range (Low_Bound) is specified as MIN_VAL, the repeated SERE describes at most as many repetitions as the high value of the range. If no range is specified, the repeated SERE describes exactly one repetition, i.e., behavior in which the Boolean expression holds exactly once (only at the last cycle on the path).

Restrictions

If the repeated SERE contains a Count, it shall be a statically computable, positive Count (i.e., indicating at least one repetition). If the Count is a Range, then each bound of the Range shall be statically computable, and the low bound of the Range shall be less than or equal to the high bound of the Range.

Informal semantics

For Boolean `A` and numbers `n` and `m`:

— `A[->n]` holds tightly on a path iff A occurs exactly n times along the path and the last cycle at which it occurs is the last cycle of the path.
— `A[->n:m]` holds tightly on a path iff A occurs between n and m times, inclusive, along the path, and the last cycle at which it occurs is the last cycle of the path.
— `A[->1:m]` holds tightly on a path iff A occurs m times or less along the path and the last cycle at which it occurs is the last cycle of the path.
— `A[->n:inf]` holds tightly on a path iff A occurs at least n times along the path and the last cycle at which it occurs is the last cycle of the path.
— `A[->1:inf]` holds tightly on a path iff A occurs one or more times along the path and the last cycle at which it occurs is the last cycle of the path.
— `A[->]` holds tightly on a path iff A occurs in the last cycle of the path and in no cycle before that.

NOTE—If a repeated SERE begins with a Sequence that is itself a repeated SERE (e.g., a[->2][*3], where the repetition operator [*3] applies to the Sequence that is itself the repeated SERE a[->2]), the semantics are the same as if that Sequence were braced (e.g., {a[->2]}[*3]).

6.1.2.4 Braced SERE

A SERE enclosed in braces is another form of sequence, as shown in Box 36.[3]

```
Braced_SERE ::=
        { SERE }
```

Box 36—Braced SERE

6.1.2.5 Clocked SERE (@)

The *SERE clock* operator **(@)**, shown in Box 37, provides a way to clock a SERE.

```
Clocked_SERE ::=
        Braced_SERE @ Clock_Expression
```

Box 37—SERE clock operator

The first operand is the braced SERE to be clocked. The second operand is a clock expression (see section 5.3) with which to clock the SERE.

Restrictions

None.

Informal semantics

For unclocked SERE A and Boolean CLK:

> A@CLK holds tightly on a given path iff (if and only if) CLK holds in the last cycle of the given path, and A holds tightly on the path obtained by extracting from the given path exactly those cycles in which CLK holds.

NOTE—When clocks are nested, the inner clock takes precedence over the outer clock. That is, the SERE {a;b@clk2;c}@clk is equivalent to the SERE {a@clk; b@clk2; c@clk}, with the outer clock applied to only the unclocked sub-SEREs. In particular, there is no conjunction of nested clocks involved.

[3]In the Verilog flavor, if a series of tokens matching **{** HDL_or_PSL_Expression **}** appears where a Sequence is allowed, then it should be interpreted as a Sequence, not as a concatenation of one argument.

Examples

Example 1

Consider the following behavior of Booleans a, b, and clk, where "time" is at the granularity observed by the verification tool:

```
time  0  1  2  3  4
-------------------
clk   0  1  0  1  0
a     0  1  1  0  0
b     0  0  0  1  0
```

The unclocked SERE {a;b} holds tightly from time 2 to time 3. It does not hold tightly over any other interval of the given behavior.

The clocked SERE {a;b}@clk holds tightly from time 0 to time 3, and also from time 1 to time 3. It does not hold tightly over any other interval of the given behavior.

Example 2

Consider the following behavior of Booleans a, b, c, clk1, and clk2, where "time" is at the granularity observed by the verification tool:

```
time  0  1  2  3  4  5  6  7
----------------------------
clk1  0  1  0  1  0  1  0  1
a     0  1  1  0  0  0  0  0
b     0  0  0  1  0  0  0  0
c     0  0  0  0  1  0  1  0
clk2  1  0  0  1  0  0  1  0
```

The unclocked SERE {{a;b};c} holds tightly from time 2 to time 4. It does not hold tightly over any other interval of the given behavior.

The multiply-clocked SERE {{a;b}@clk1;c}@clk2 holds tightly from time 0 to time 6 and from time 1 to time 6. It does not hold tightly over any other interval of the given behavior.

The singly-clocked SEREs {{a;b};c}@clk1 and {{a;b};c}@clk2 do not hold tightly over any interval of the given behavior.

6.1.3 Named sequences

A given sequence may describe behavior that can occur in different contexts (i.e., in conjunction with other behavior). In such a case, it is convenient to be able to define the sequence once and refer to the single definition in each context in which the sequence applies. Declaration and instantiation of *named sequences* provide this capability.

6.1.3.1 Sequence declaration

A *sequence declaration*, shown in Box 38, defines a sequence and gives it a name. A sequence declaration can also specify a list of formal parameters that can be referenced within the sequence.

```
PSL_Declaration ::=
        Sequence_Declaration
Sequence_Declaration ::=
        sequence PSL_Identifier [ ( Formal_Parameter_List ) ] DEF_SYM Sequence ;
Formal_Parameter_List ::=
        Formal_Parameter { ; Formal_Parameter }
Formal_Parameter ::=
        sequence_Param_Type PSL_Identifier { , PSL_Identifier }
sequence_Param_Type ::=
        const | boolean | sequence
```

Box 38—Sequence declaration

Informal Semantics

The PSL identifier following the keyword `sequence` in the sequence declaration is the name of the sequence. The PSL identifiers given in the formal parameter list are the names of the formal parameters of the named sequence.

Restrictions

The name of a declared sequence shall not be the same as the name of any other PSL declaration in the same verification unit. Formal parameters of a sequence declaration are limited to parameter kinds `const`, `boolean`, and `sequence`.

Examples

```
sequence BusArb (boolean br, bg; const n) =
                        { br; (br && !bg)[*0:n]; br && bg };
```

The named sequence `BusArb` represents a generic bus arbitration sequence involving formal parameters `br` (bus request) and `bg` (bus grant), as well as a parameter `n` that specifies the maximum delay in receiving the bus grant.

```
sequence ReadCycle (sequence ba; boolean bb, ar, dr) =
                        { ba; {bb[*]} && {ar[->]; dr[->]}; !bb };
```

The named sequence `ReadCycle` represents a generic read operation involving a bus arbitration sequence and Boolean conditions `bb` (bus busy), `ar` (address ready), and `dr` (data ready).

NOTE—There is no requirement to use formal parameters in a sequence declaration. A declared sequence may refer directly to signals in the design as well as to formal parameters.

6.1.3.2 Sequence instantiation

A *sequence instantiation*, shown in Box 39, creates an instance of a named sequence and provides actual parameters for formal parameters (if any) of the named sequence.

```
Sequence_Instance ::=
        sequence_Name [ ( Actual_Parameter_List ) ]
Actual_Parameter_List ::=
        sequence_Actual_Parameter { , sequence_Actual_Parameter }
sequence_Actual_Parameter ::=
        Number | Boolean | Sequence
```

Box 39—Sequence instantiation

Restrictions

For each formal parameter of the named sequence, the sequence instantiation shall provide a corresponding actual parameter. For a `const` formal parameter, the actual parameter shall be a statically evaluable integer expression. For a `boolean` formal parameter, the actual parameter shall be a Boolean expression. For a `sequence` formal parameter, the actual parameter shall be a Sequence.

Informal semantics

An instance of a named sequence describes the behavior that is described by the sequence obtained from the named sequence by replacing each formal parameter in the named sequence with the corresponding actual parameter from the sequence instantiation.

Examples

Given the declarations for the sequences `BusArb` and `ReadCycle` in 6.1.3.1,

```
BusArb (breq, back, 3)
```

is equivalent to

```
{ breq; (breq && !back)[*0:3]; breq && back }
```

and

```
ReadCycle(BusArb(breq, back, 5), breq, ardy, drdy)
```

is equivalent to

```
{ { breq; (breq && !back)[*0:5]; breq && back }; {breq[*]} && {ardy[->];
drdy[->]}; !breq }
```

6.1.4 Named endpoints

An *endpoint* is a special kind of Boolean-valued variable that indicates when an associated sequence completes.

6.1.4.1 Endpoint declaration

An *endpoint declaration*, shown in Box 40, defines an endpoint for a given sequence and gives the endpoint a name. An endpoint declaration can also specify a list of formal parameters that can be referenced within the sequence.

```
PSL_Declaration ::=
        Endpoint_Declaration
Endpoint_Declaration ::=
        endpoint PSL_Identifier [ ( Formal_Parameter_List ) ] DEF_SYM Sequence ;
Formal_Parameter_List ::=
        Formal_Parameter { ; Formal_Parameter }
Formal_Parameter ::=
        sequence_Param_Type PSL_Identifier { , PSL_Identifier }
sequence_Param_Type ::=
        const | boolean | sequence
```

Box 40—Endpoint declaration

Informal Semantics

The PSL identifier following the keyword `endpoint` in the endpoint declaration is the name of the endpoint. The PSL identifiers given in the formal parameter list are the names of the formal parameters of the named endpoint.

Restrictions

The name of a declared endpoint shall not be the same as the name of any other PSL declaration in the same verification unit. Formal parameters of an endpoint declaration are limited to parameter kinds `const`, `boolean`, and `sequence`.

Example

```
endpoint ActiveLowReset (boolean rb, clk; const n) =
               { rb!=1'b1[*n:inf]; rb==1'b1 } @(posedge clk);
```

The endpoint `ActiveLowReset` represents a generic reset sequence in which the reset signal is asserted (set to `0`) for at least `n` cycles of the relevant clock before being released (set to `1`).

NOTE—There is no requirement to use formal parameters in an endpoint declaration. The sequence in an endpoint declaration may refer directly to signals in the design as well as to formal parameters.

6.1.4.2 Endpoint instantiation

An *endpoint instantiation*, shown in Box 41, creates an instance of a named endpoint and provides actual parameters for formal parameters (if any) of the named endpoint.

```
Boolean ::=
        boolean_HDL_or_PSL_Expression
boolean_HDL_or_PSL_Expression ::=
        endpoint_Name [ ( Actual_Parameter_List ) ]
Actual_Parameter_List ::=
        sequence_Actual_Parameter { , sequence_Actual_Parameter }
sequence_Actual_Parameter ::=
        Number | Boolean | Sequence
```

Box 41—Endpoint instantiation

Restrictions

For each formal parameter of the named endpoint, the endpoint instantiation shall provide a corresponding actual parameter. For a `const` formal parameter, the actual parameter shall be a statically evaluable integer expression. For a `boolean` formal parameter, the actual parameter shall be a Boolean expression. For a `sequence` formal parameter, the actual parameter shall be a Sequence.

Informal semantics

An instance of a named endpoint has the value *True* in any evaluation cycle that is the last cycle of a behavior on which the associated sequence, modified by replacing each formal parameter in the named sequence with the corresponding actual parameter from the sequence instantiation, holds tightly.

Examples

Given the declaration for the endpoint `ActiveLowReset` in 6.1.4.1,

```
   ActiveLowReset (res, mclk, 3)
```

is *True* each time `res` has the value `1'b1` at the rising edge of `mclk`, provided that `res` did not have the value `1'b1` at the three immediately preceding rising edges of `mclk`; it is *False* otherwise.

6.2 Properties

Properties express temporal relationships among Boolean expressions, sequential expressions, and subordinate properties. Various operators are defined to express various temporal relationships.

Some operators occur in families. A *family of operators* is a group of operators that are related. A family of operators usually share a common prefix, which is the name of the family, and optional suffixes that include, for example, the strings `!`, `_`, and `!_`. For instance, the until family of operators include the operators `until!`, `until`, `until!_`, and `until_`.

6.2.1 FL properties

FL Properties, shown in Box 42, describe single- or multi-cycle behavior built from Boolean expressions, sequential expressions, and subordinate properties.

```
FL_Property ::=
        Boolean
        | ( FL_Property )
```

Box 42—FL properties

The most basic FL Property is a Boolean expression. An FL Property enclosed in parentheses is also an FL property.

More complex FL properties are built from Boolean expressions, sequential expressions, and subordinate properties using various temporal operators.

NOTE—Like Boolean expressions, FL properties are grouped using parentheses (`()`), as opposed to SEREs that are grouped using curly braces (`{}`).

6.2.1.1 Sequential FL properties

Sequential expressions are FL properties which describe a demand that a certain single- or multi-cycle behavior (built from Boolean expressions) hold.

```
FL_Property ::=
        Sequence [ ! ]
```

Box 43—Sequential FL property

Restrictions

None.

Informal semantics

For a Sequence S:

- The FL property S! holds on a given path iff there exists a prefix of the path on which S holds tightly.
- The FL property S holds on a given path iff either there exists a prefix of the path on which S holds tightly, or the given path exhibits no evidence that the property S! does not hold on it.

NOTE—If S contains no contradictions, an easier description of the semantics of the property S can be given as follows: The FL property S holds on a given path iff either there exists a prefix of the path on which S holds tightly, or the given path can be extended to a path on which S holds tightly.

6.2.1.2 Clocked FL properties

The *FL clock operator* operator (`@`), shown in Box 44, provides a way to clock an FL Property.

```
FL_Property ::=
        FL_Property @ Clock_Expression
```

Box 44—FL property clock operator

The first operand is the FL Property to be clocked. The second operand is a Boolean expression with which to clock the FL Property.

Restrictions

None.

Informal semantics

For FL property `A` and Boolean `CLK`:

> `A@CLK` holds on a given path iff `A` holds on the path obtained by extracting from the given path exactly those cycles in which `CLK` holds.

NOTE—When clocks are nested, the inner clock takes precedence over the outer clock. That is, the property `(a -> b@clk2)@clk` is equivalent to the property `(a@clk -> b@clk2)`, with the outer clock applied to only the unclocked sub-properties (if any). In particular, there is no conjunction of nested clocks involved.

Example 1

Consider the following behavior of Booleans `a`, `b`, and `clk`, where "time" is at the granularity observed by the verification tool:

```
time  0  1  2  3  4  5  6  7  8  9
----------------------------------
clk   0  1  0  1  0  1  0  1  0  1
a     0  0  0  1  1  1  0  0  0  0
b     0  0  0  0  0  1  0  1  1  0
```

The unclocked FL Property

```
(a until! b)
```

holds at times 5, 7, and 8, because `b` holds at each of those times. The property also holds at times 3 and 4, because `a` holds at those times and continues to hold until `b` holds at time 5. It does not hold at any other time of the given behavior.

The clocked FL Property

```
(a until! b) @clk
```

holds at times 2, 3, 4, 5, 6, and 7. It does not hold at any other time of the given behavior.

Example 2

Consider the following behavior of Booleans `a`, `b`, `c`, `clk1`, and `clk2`, where "time" is at the granularity observed by the verification tool:

```
time  0  1  2  3  4  5  6  7  8  9
----------------------------------
clk1  0  1  0  1  0  1  0  1  0  1
a     0  0  0  1  1  1  0  0  0  0
b     0  0  0  0  0  1  0  1  1  0
c     1  0  0  0  0  1  1  0  0  0
clk2  1  0  0  1  0  0  1  0  0  1
```

The unclocked FL Property

```
(c && next! (a until! b))
```

holds at time 6. It does not hold at any other time of the given behavior.

The singly-clocked FL Property

```
(c && next! (a until! b))@clk1
```

holds at times 4 and 5. It does not hold at any other time of the given behavior.

The singly-clocked FL Property

```
(a until! b)@clk2
```

does not hold at any time of the given behavior.

The multiply-clocked FL Property

```
(c && next! (a until! b)@clk1)@clk2
```

holds at time 0. It does not hold at any other time of the given behavior.

6.2.1.3 Simple FL properties

6.2.1.3.1 always

The `always` operator, shown in Box 45, specifies that an FL property holds at all times, starting from the present.

FL_Property ::=
        always FL_Property

Box 45—always operator

The operand of the `always` operator is an FL Property.

Restrictions

None.

Informal semantics

An `always` property holds in the current cycle of a given path iff the FL Property that is the operand holds at the current cycle and all subsequent cycles.

NOTE—If the operand (FL property) is *temporal* (i.e., spans more than one cycle), then the `always` operator defines a property that describes overlapping occurrences of the behavior described by the operand. For example, the property always `{a;b;c}` describes any behavior in which `{a;b;c}` holds in every cycle, thus any behavior in which `a` holds in the first and every subsequent cycle, `b` holds in the second and every subsequent cycle, and `c` holds in the third and every subsequent cycle.

6.2.1.3.2 never

The `never` operator, shown in Box 46, specifies that an FL property or a sequence never holds.

```
FL_Property ::=
        never FL_Property
```

Box 46—never operator

The operand of the `never` operator is an FL Property.

Restrictions

Within the simple subset (see section 4.4.4), the operand of a `never` property is restricted to be a Boolean expression or a sequence.

Informal semantics

A `never` property holds in the current cycle of a given path iff the FL Property that is the operand does not hold at the current cycle and does not hold at any future cycle.

6.2.1.3.3 eventually!

The `eventually!` operator, shown in Box 47, specifies that an FL property holds at the current cycle or at some future cycle.

```
FL_Property ::=
        eventually! FL_Property
```

Box 47—eventually! operator

The operand of the `eventually!` operator is an FL Property.

Restrictions

Within the simple subset (see section 4.4.4), the operand of an `eventually!` property is restricted to be a Boolean or a Sequence.

Informal semantics

An `eventually!` property holds in the current cycle of a given path iff the FL Property that is the operand holds at the current cycle or at some future cycle.

6.2.1.3.4 next

The `next` family of operators, shown in Box 48, specify that an FL property holds at some next cycle.

```
FL_Property ::=
        next! FL_Property
      | next FL_Property
      | next! [ Number ] (FL_Property)
      | next [ Number ] (FL_Property)
```

Box 48—next operators

The FL Property that is the operand of the `next!` or `next` operator is a property that holds at some next cycle. If present, the Number indicates at which next cycle the property holds, that is, for number *i*, the property holds at the *i*th next cycle. If the Number operand is omitted, the property holds at the very next cycle.

The `next!` operator is a strong operator, thus it specifies that there is a next cycle (and so does not hold at the last cycle, no matter what the operand). Similarly, `next![i]` specifies that there are at least *i* next cycles.

The `next` operator is a weak operator, thus it does not specifies that there is a next cycle, only that if there is, the property that is the operand holds. Thus, a weak next property holds at the last cycle of a finite behavior, no matter what the operand. Similarly, `next[i]` does not specify that there are at least *i* next cycles.

NOTE—The Number may be 0. That is, `next[0](f)` is allowed, which says that `f` holds at the current cycle.

Restrictions

If a property contains a Number, then the Number shall be statically computable.

Informal semantics

— A `next!` property holds in the current cycle of a given path iff:
  1) there is a next cycle and
  2) the FL property that is the operand holds at the next cycle.
— A `next` property holds in the current cycle of a given path iff:
  1) there is not a next cycle or
  2) the FL property that is the operand holds at the next cycle.
— A `next![i]` property holds in the current cycle of a given path iff:
  1) there is an *i*th next cycle and
  2) the FL property that is the operand holds at the *i*th next cycle.
— A `next[i]` property holds in the current cycle of a given path iff:
  1) there is not an *i*th next cycle or
  2) the FL property that is the operand holds at the *i*th next cycle.

NOTE—The formula `next(f)` is equivalent to the formula `next[1](f)`.

6.2.1.4 Extended next FL properties

6.2.1.4.1 next_a

The `next_a` family of operators, shown in Box 49, specify that an FL property holds at all cycles of a range of future cycles.

```
FL_Property ::=
        next_a! [ finite_Range ] ( FL_Property )
      | next_a [ finite_Range ] ( FL_Property )
```

Box 49—next_a operators

The FL Property that is the operand of the `next_a!` or `next_a` operator is a property that holds at all cycles between the i^{th} and j^{th} next cycles, inclusive, where *i* and *j* are the low and high bounds, respectively, of the finite Range.

The `next_a!` operator is a strong operator, thus it specifies that there is a j^{th} next cycle, where *j* is the high bound of the Range.

The `next_a` operator is a weak operator, thus it does not specify that any of the i^{th} through j^{th} next cycles necessarily exist.

Restrictions

If a `next_a` or `next_a!` property contains a Range, then the Range shall be a finite Range, each bound of the Range shall be statically computable, and the left bound of the Range shall be less than or equal to the right bound of the Range.

Informal semantics

— A `next_a![i:j]` property holds in the current cycle of a given path iff:
  1) there is a j^{th} next cycle and
  2) the FL Property that is the operand holds at all cycles between the i^{th} and j^{th} next cycle, inclusive.

— A `next_a[i:j]` property holds in the current cycle of a given path iff the FL Property that is the operand holds at all cycles between the i^{th} and j^{th} next cycle, inclusive. (If not all those cycles exist, then the FL Property that is the operand holds on as many as do exist).

NOTE—The left bound of the Range may be 0. For example, `next_a[0:n](f)` is allowed, which says that `f` holds starting in the current cycle, and for *n* cycles following the current cycle.

6.2.1.4.2 next_e

The `next_e` family of operators, shown in Box 50, specify that an FL property holds at least once within some range of future cycles.

FL_Property ::=
    next_e! [*finite*_Range **] (** FL_Property **)**
  | **next_e [** *finite*_Range **] (** FL_Property **)**

Box 50—next_e operators

The FL Property that is the operand of the `next_e!` or `next_e` operator is a property that holds at least once between the *i*th and *j*th next cycle, inclusive, where *i* and *j* are the low and high bounds, respectively, of the finite Range.

The `next_e!` operator is a strong operator, thus it specifies that there are enough cycles so the FL property that is the operand has a chance to hold.

The `next_e` operator is a weak operator, thus it does not specify that there are enough cycles so the FL property that is the operand has a chance to hold.

Restrictions

If a next_e or next_e! property contains a Range, then the Range shall be a finite Range, each bound of the Range shall be statically computable, and the left bound of the Range shall be less than or equal to the right bound of the Range.

Informal semantics

— A `next_e![i..j]` property holds in the current cycle of a given path iff there is some cycle between the *i*th and *j*th next cycle, inclusive, where the FL Property that is the operand holds.
— A `next_e[i..j]` property holds in the current cycle of a given path iff
  1) there are less than j next cycles following the current cycle, or
  2) there is some cycle between the *i*th and *j*th next cycle, inclusive, where the FL Property that is the operand holds.

NOTE—The left bound of the Range may be 0. For example, `next_e[0:n](f)` is allowed, which says that `f` holds either in the current cycle or in one of the *n* cycles following the current cycle.

6.2.1.4.3 next_event

The `next_event` family of operators, shown in Box 51, specify that an FL property holds at the next occurrence of a Boolean expression. The next occurrence of the Boolean expression includes an occurrence at the current cycle..

FL_Property ::=
    next_event! (Boolean **) (** FL_Property **)**
  | **next_event (** Boolean **) (** FL_Property **)**
  | **next_event! (** Boolean **) [** *positive*_Number **] (** FL_Property **)**
  | **next_event (** Boolean **) [** *positive*_Number **] (** FL_Property **)**

Box 51—next_event operators

The rightmost operand of the `next_event!` or `next_event` operator is an FL Property that holds at the next occurrence of the leftmost operand. If the FL Property includes a Number, then the property holds at the i^{th} occurrence of the leftmost operand (where *i* is the value of the Number), rather than at the very next occurrence.

The `next_event!` operator is a strong operator, thus it specifies that there is a next occurrence of the leftmost operand. Similarly, `next_event![i]` specifies that there are at least *i* occurrences.

The `next_event` operator is a weak operator, thus it does not specify that there is a next occurrence of the leftmost operand. Similarly, `next_event[i]` does not specify that there are at least *i* next occurrences.

Restrictions

If a `next_event` or `next_event!` property contains a Number, then the Number shall be a statically computable, positive Number.

Informal semantics

— A `next_event!` property holds in the current cycle of a given path iff:
1) the Boolean expression and the FL Property that are the operands both hold at the current cycle, or at some future cycle, and
2) the Boolean expression holds at some future cycle, and the FL property that is the operand holds at the next cycle in which the Boolean expression holds.

— A `next_event` property holds in the current cycle of a given path iff:
1) the Boolean expression that is the operand does not hold at the current cycle, nor does it hold at any future cycle; or
2) the Boolean expression that is the operand holds at the current cycle or at some future cycle, and the FL property that is the operand holds at the next cycle in which the Boolean expression holds.

— A `next_event![i]` property holds in the current cycle of a given path iff:
1) the Boolean expression that is the operand holds at least *i* times, starting at the current cycle, and
2) the FL property that is the operand holds at the i^{th} occurrence of the Boolean expression.

— A `next_event[i]` property holds in the current cycle of a given path iff:
1) the Boolean expression that is the operand does not hold at least *i* times, starting at the current cycle, or
2) the Boolean expression that is the operand holds at least *i* times, starting at the current cycle, and the FL property that is the operand holds at the i^{th} occurrence of the Boolean expression.

NOTE—The formula `next_event(true)(f)` is equivalent to the formula `next[0](f)`. Similarly, if `p` holds in the current cycle, then `next_event(p)(f)` is equivalent to `next_event(true)(f)` and therefore to `next[0](f)`. However, none of these is equivalent to `next(f)`.

6.2.1.4.4 next_event_a

The `next_event_a` family of operators, shown in Box 52, specify that an FL property holds at a range of the next occurrences of a Boolean expression. The next occurrences of the Boolean expression include an occurrence at the current cycle.

```
FL_Property ::=
        next_event_a! ( Boolean ) [ finite_positive_Range ] ( FL_Property )
      | next_event_a ( Boolean ) [ finite_positive_Range ] ( FL_Property )
```

Box 52—next_event_a operators

The rightmost operand of the `next_event_a!` or `next_event_a` operator is an FL Property that holds at the specified `Range` of next occurrences of the Boolean expression that is the leftmost operand. The FL Property that is the rightmost operand holds on the i^{th} through j^{th} occurrences (inclusive) of the Boolean expression, where i and j are the low and high bounds, respectively, of the Range.

The `next_event_a!` operator is a strong operator, thus it specifies that there are at least j occurrences of the leftmost operand.

The `next_event_a` operator is a weak operator, thus it does not specify that there are j occurrences of the left-most operand.

Restrictions

If a `next_event_a` or `next_event_a!` property contains a Range, then the Range shall be a finite, positive Range, each bound of the Range shall be statically computable, and the left bound of the Range shall be less than or equal to the right bound of the Range

Informal semantics

— A `next_event_a![i..j]` property holds in the current cycle of a given path iff:
  1) the Boolean expression that is the operand holds at least j times, starting at the current cycle, and
  2) the FL property that is the operand holds at the i^{th} through j^{th} occurrences, inclusive, of the Boolean expression.

— A `next_event_a[i..j]` property holds in a given cycle of a given path iff the FL property that is the operand holds at the i^{th} through j^{th} occurrences, inclusive, of the Boolean expression, starting at the current cycle. If there are less than j occurrences of the Boolean expression, then the FL property that is the operand holds on all of them, starting from the i^{th} occurrence.

.

6.2.1.4.5 next_event_e

The `next_event_e` family of operators, shown in Box 53, specify that an FL property holds at least once during a range of next occurrences of a Boolean expression. The next occurrences of the Boolean expression include an occurrence at the current cycle.

```
FL_Property ::=
        next_event_e! ( Boolean ) [ finite_positive_Range ] ( FL_Property )
      | next_event_e ( Boolean ) [ finite_positive_Range ] ( FL_Property )
```

Box 53—next_event_e operators

The rightmost operand of the `next_event_e!` or `next_event_e` operator is an FL Property that holds at least once during the specified `Range` of next occurrences of the Boolean expression that is the leftmost operand. The FL Property that is the rightmost operand holds on one of the i^{th} through j^{th} occurrences (inclusive) of the Boolean expression, where *i* and *j* are the low and high bounds, respectively, of the Range.

The `next_event_e!` operator is a strong operator, thus it specifies that there are enough cycles so that the FL Property has a chance to hold.

The `next_event_e` operator is a weak operator, thus it does not specify that there are enough cycles so that the FL Property has a chance to hold.

Restrictions

If a `next_event_e` or `next_event_e!` property contains a Range, then the Range shall be a finite, positive Range, each bound of the Range shall be statically computable, and the left bound of the Range shall be less than or equal to the right bound of the Range.

Informal semantics

— A `next_event_e![i..j]` property holds in the current cycle of a given path iff there is some cycle during the i^{th} through j^{th} next occurrences of the Boolean expression at which the FL Property that is the operand holds.
— A `next_event_e[i..j]` property holds in the current cycle of a given path iff:
  1) there are less than *j* next occurrences of the Boolean expression or
  2) there is some cycle during the i^{th} through j^{th} next occurrences of the Boolean expression at which the FL Property that is the operand holds.

6.2.1.5 Compound FL properties

6.2.1.5.1 abort

The `abort` operator, shown in Box 54, specifies a condition that removes any obligation for a given FL property to hold.

```
FL_Property ::=
        FL_Property abort Boolean
```

Box 54—abort operator

The left operand of the `abort` operator is the FL Property to be aborted. The right operand of the `abort` operator is the Boolean condition that causes the abort to occur.

Restrictions

None.

Informal semantics

An `abort` property holds in the current cycle of a given path iff:

— the FL Property that is the left operand holds, or
— the series of cycles starting from the current cycle and ending with the cycle in which the Boolean condition that is the right operand holds does not contradict the FL Property that is the left operand.

Example

Using `abort` to model an asynchronous interrupt: "A request is always followed by an acknowledge, unless a cancellation occurs."

```
   always ((req -> eventually! ack) abort cancel);
```

6.2.1.5.2 before

The `before` family of operators, shown in Box 55, specify that one FL property holds before a second FL property holds.

```
FL_Property ::=
      FL_Property before! FL_Property
    | FL_Property before!_ FL_Property
    | FL_Property before FL_Property
    | FL_Property before_ FL_Property
```

Box 55—before operators

The left operand of the `before` family of operators is an FL Property that holds before the FL Property that is the right operand holds.

The `before!` and `before!_` operators are strong operators, thus they specify that the left FL Property eventually holds.

The `before` and `before_` operators are weak operators, thus they do not specify that the left FL Property eventually holds.

The `before!` and `before` operators are non-inclusive operators, that is, they specify that the left operand holds strictly before the right operand holds.

The `before!_` and `before_` operators are inclusive operators, that is, they specify that the left operand holds before or at the same cycle as the right operand holds.

Restrictions

Within the simple subset (see section 4.4.4), each operand of a `before` property is restricted to be a Boolean expression.

Informal semantics

- A `before!` property holds in the current cycle of a given path iff:
  1) the FL Property that is the left operand holds at the current cycle or at some future cycle and
  2) the FL Property that is the left operand holds strictly before the FL Property that is the right operand holds, or the right operand never holds.
- A `before!_` property holds in the current cycle of a given path iff:
  1) the FL Property that is the left operand holds at the current cycle or at some future cycle and
  2) the FL Property that is the left operand holds before or at the same cycle as the FL Property that is the right operand, or the right operand never holds.
- A `before` property holds in the current cycle of a given path iff:
  1) neither the FL Property that is the left operand nor the FL Property that is the right operand ever hold in any future cycle; or
  2) the FL Property that is the left operand holds strictly before the FL Property that is the right operand holds.
- A `before_` property holds in the current cycle of a given path iff:
  1) neither the FL Property that is the left operand nor the FL Property that is the right operand ever hold in any future cycle; or
  2) the FL Property that is the left operand holds before or at the same cycle as the FL Property that is the right operand.

6.2.1.5.3 until

The `until` family of operators, shown in Box 56, specify that one FL property holds until a second FL property holds.

```
FL_Property ::=
          FL_Property  until! FL_Property
        | FL_Property  until!_ FL_Property
        | FL_Property  until FL_Property
        | FL_Property  until_ FL_Property
```

Box 56—until operators

The left operand of the `until` family of operators is an FL Property that holds until the FL Property that is the right operand holds. The right operand is called the "terminating property".

The `until!` and `until!_` operators are strong operators, thus they specify that the terminating property eventually holds.

The `until` and `until_` operators are weak operators, thus they do not specify that the terminating property eventually holds (and if it does not eventually hold, then the FL Property that is the left operand holds forever).

The `until!` and `until` operators are non-inclusive operators, that is, they specify that the left operand holds up to, but not necessarily including, the cycle in which the right operand holds.

The `until!_` and `until_` operators are inclusive operators, that is, they specify that the left operand holds up to and including the cycle in which the right operand holds.

Restrictions

Within the simple subset (see section 4.4.4), the right operand of an `until!` or `until` property is restricted to be a Boolean expression, and both the left and right operands of an `until!_` or `until_` property are restricted to be a Boolean expression.

Informal semantics

— An `until!` property holds in the current cycle of a given path iff:
  1) the FL Property that is the right operand holds at the current cycle or at some future cycle; and
  2) the FL Property that is the left operand holds at all cycles up to, but not necessarily including, the earliest cycle at which the FL Property that is the right operand holds.

— An `until!_` property holds in the current cycle of a given path iff:
  1) the FL Property that is the right operand holds at the current cycle or at some future cycle and
  2) the FL Property that is the left operand holds at all cycles up to and including the earliest cycle at which the FL Property that is the right operand holds.

— An `until` property holds in the current cycle of a given path iff:
  1) the FL Property that is the left operand holds forever; or
  2) the FL Property that is the right operand holds at the current cycle or at some future cycle, and the FL Property that is the left operand holds at all cycles up to, but not necessarily including, the earliest cycle at which the FL Property that is the right operand holds.

— An `until_` property holds in the current cycle of a given path iff:
  1) the FL Property that is the left operand holds forever or
  2) the FL Property that is the right operand holds at the current cycle or at some future cycle, and the FL Property that is the left operand holds at all cycles up to and including the earliest cycle at which the FL Property that is the right operand holds.

6.2.1.6 Sequence-based FL properties

6.2.1.6.1 Suffix implication

The *suffix implication* family of operators, shown in Box 57, specify that an FL property or sequence holds if some pre-requisite sequence holds.

```
FL_Property ::=
        { Sequence } ( FL_Property )
      | Sequence |-> FL_Property
      | Sequence |=> FL_Property
```

Box 57—Suffix implication operators

The right operand of the operators is an FL property that is specified to hold if the Sequence that is the left operand holds.

Restrictions

None.

Informal semantics

- A Sequence |-> FL_Property holds in a given cycle of a given path iff:
  1) the Sequence that is the left operand does not hold at the give cycle; or
  2) the FL Property that is the right operand holds in any cycle C such that the Sequence that is the left operand holds tightly from the given cycle to C.
- A Sequence |=> FL_Property holds in a given cycle of a given path iff:
  1) the Sequence that is the left operand does not hold at the given cycle; or
  2) the FL Property that is the right operand holds in the cycle immeditately after any cycle C such that the Sequence that is the left operand holds tightly from the given cycle to C.

NOTE—A {Sequence}(FL_Property) FL property has the same semantics as Sequence |-> FL_Property.

6.2.1.7 Logical FL properties

6.2.1.7.1 Logical implication

The *logical implication* operator (->), shown in Box 58, is used to specify logical implication.

```
FL_Property ::=
        FL_Property  -> FL_Property
```

Box 58—Logical implication operator

The right operand of the logical implication operator is an FL Property that is specified to hold if the FL Property that is the left operand holds.

Restrictions

Within the simple subset (see section 4.4.4), the left operand of a logical implication property is restricted to be a Boolean expression.

Informal semantics

A logical implication property holds in a given cycle of a given path iff:

- the FL Property that is the left operand does not hold at the given cycle or
- the FL Property that is the right operand does hold at the given cycle.

6.2.1.7.2 Logical iff

The *logical iff* operator (<->), shown in Box 59, is used to specify the iff (if and only if) relation between two properties.

Box 59—Logical iff operator

The two operands of the logical iff operator are FL Properties. The logical iff operator specifies that either both operands hold, or neither operand holds.

```
FL_Property ::=
        FL_Property <-> FL_Property
```

Restrictions

Within the simple subset (see section 4.4.4), both operands of a logical iff property are restricted to be a Boolean expression.

Informal semantics

A logical iff property holds in a given cycle of a given path iff:

— both FL Properties that are operands hold at the given cycle or
— neither of the FL Properties that are operands holds at the given cycle.

6.2.1.7.3 Logical and

The *logical and* operator, shown in Box 60, is used to specify logical and.

```
FL_Property ::=
        FL_Property AND_OP FL_Property
```

Box 60—Logical and operator

The operands of the logical and operator are two FL Properties that are both specified to hold.

Restrictions

Within the simple subset (see section 4.4.4), the left operand of a logical and property is restricted to be a Boolean expression.

Informal semantics

A logical and property holds in a given cycle of a given path iff the FL Properties that are the operands both hold at the given cycle.

6.2.1.7.4 Logical or

The *logical or* operator, shown in Box 61, is used to specify logical or.

```
FL_Property ::=
        FL_Property OR_OP FL_Property
```

Box 61—Logical or operator

The operands of the logical or operator are two FL Properties, at least one of which is specified to hold.

Restrictions

Within the simple subset (see section 4.4.4), the left operand of a logical or property is restricted to be a Boolean expression.

Informal semantics

A logical or property holds in a given cycle of a given path iff at least one of the FL Properties that are the operands holds at the given cycle.

6.2.1.7.5 Logical not

The *logical not* operator, shown in Box 62, is used to specify logical negation.

```
FL_Property ::=
        NOT_OP FL_Property
```

Box 62—Logical not operator

The operand of the logical not operator is an FL Property that is specified to not hold.

Restrictions

Within the simple subset (see section 4.4.4), the operand of a logical not property is restricted to be a Boolean expression.

Informal semantics

A logical not property holds in a given cycle of a given path iff the FL Property that is the operand does not hold at the given cycle.

6.2.1.8 LTL operators

The *LTL operators*, shown in Box 63, provide standard LTL syntax for other PSL operators.

```
FL_Property ::=
          X FL_Property
        | X! FL_Property
        | F FL_Property
        | G FL_Property
        | [ FL_Property U FL_Property ]
        | [ FL_Property W FL_Property ]
```

Box 63—LTL operators

The standard LTL operators are alternate syntax for the equivalent PSL operators, as shown in Table 4.

Table 4—PSL equivalents

Standard LTL operator	**Equivalent PSL operator**
`X`	`next`
`X!`	`next!`
`F`	`eventually!`
`G`	`always`
`U`	`until!`
`W`	`until`

Restrictions

The restrictions that apply to each equivalent PSL operator also apply to the corresponding standard LTL operator.

6.2.2 Optional Branching Extension (OBE) properties

Properties of the Optional Branching Extension (*OBE*), shown in Box 64, are interpreted over trees of states as opposed to properties of the Foundation Language (FL), which are interpreted over sequences of states. A *tree of states* is obtained from the model by unwrapping, where each path in the tree corresponds to some computation path of the model. A node in the tree branches to several nodes as a result of non-determinism in the model. A completely deterministic model unwraps to a tree of exactly one path, i.e., to a sequence of states. An OBE property holds or does not hold for a specific state of the tree.

```
OBE_Property ::=
        Boolean
      | ( OBE_Property )
```

Box 64—OBE properties

The most basic OBE Property is a Boolean expression. An OBE Property enclosed in parentheses is also an OBE Property.

6.2.2.1 Universal OBE properties

6.2.2.1.1 AX operator

The AX operator, shown in Box 65, specifies that an OBE property holds at all next states of the given state.

```
OBE_Property ::=
        AX OBE_Property
```

Box 65—AX operator

The operand of AX is an OBE Property that is specified to hold at all next states of the given state.

Restrictions

None.

Informal semantics

An AX property holds at a given state iff, for all paths beginning at the given state, the OBE Property that is the operand holds at the next state.

6.2.2.1.2 AG operator

The AG operator, shown in Box 66, specifies that an OBE property holds at the given state and at all future states.

```
OBE_Property ::=
        AG OBE_Property
```

Box 66—AG operator

The operand of AG is an OBE Property that is specified to hold at the given state and at all future states.

Restrictions

None.

Informal semantics

An AG property holds at a given state iff, for all paths beginning at the given state, the OBE Property that is the operand holds at the given state and at all future states.

6.2.2.1.3 AF operator

The AF operator, shown in Box 67, specifies that an OBE property holds now or at some future state, for all paths beginning at the current state.

```
OBE_Property ::=
        AF OBE_Property
```

Box 67—AF operator

The operand of AF is an OBE Property that is specified to hold now or at some future state, for all paths beginning at the current state.

Restrictions

None.

Informal semantics

An AF property holds at a given state iff, for all paths beginning at the given state, the OBE Property that is the operand holds at the first state or at some future state.

6.2.2.1.4 AU operator

The AU operator, shown in Box 68, specifies that an OBE property holds until a specified terminating property holds, for all paths beginning at the given state.

```
OBE_Property ::=
        A [ OBE_Property U OBE_Property ]
```

Box 68—AU operator

The first operand of AU is an OBE Property that is specified to hold until the OBE Property that is the second operand holds along all paths starting at the given state.

Restrictions

None.

Informal semantics

An AU property holds at a given state iff, for all paths beginning at the given state:

— the OBE Property that is the right operand holds at the current state or at some future state; and
— the OBE Property that is the left operand holds at all states, up to but not necessarily including, the state in which the OBE Property that is the right operand holds.

6.2.2.2 Existential OBE properties

6.2.2.2.1 EX operator

The EX operator, shown in Box 69, specifies that an OBE property holds at some next state.

The operand of EX is an OBE property that is specified to hold at some next state of the given state.

```
OBE_Property ::=
        EX OBE_Property
```

Box 69—EX operator

Restrictions

None.

Informal semantics

An EX property holds at a given state iff there exists a path beginning at the given state, such that the OBE Property that is the operand holds at the next state.

6.2.2.2.2 EG operator

The EG operator, shown in Box 70, specifies that an OBE property holds at the current state and at all future states of some path beginning at the current state.

```
OBE_Property ::=
        EG OBE_Property
```

Box 70—EG operator

The operand of EG is an OBE Property that is specified to hold at the current state and at all future states of some path beginning at the given state.

Restrictions

None.

Informal semantics

An EG property holds at a given state iff there exists a path beginning at the given state, such that the OBE Property that is the operand holds at the given state and at all future states.

6.2.2.2.3 EF operator

The EF operator, shown in Box 71, specifies that an OBE property holds now or at some future state of some path beginning at the given state.

```
OBE_Property ::=
        EF OBE_Property
```

Box 71—EF operator

The operand of EF is an OBE Property that is specified to hold now or at some future state of some path beginning at the given state.

Restrictions

None.

Informal semantics

An EF property holds at a given state iff there exists a path beginning at the given state, such that the OBE Property that is the operand holds at the current state or at some future state.

6.2.2.2.4 EU operator

The EU operator, shown in Box 72, specifies that an OBE property holds until a specified terminating property holds, for some path beginning at the given state.

```
OBE_Property ::=
        E [ OBE_Property U OBE_Property ]
```

Box 72—EU operator

The first operand of EU is an OBE Property that is specified to hold until the OBE Property that is the second operand holds for some path beginning at the given state.

Restrictions

None.

Informal semantics

An EU property holds at a given state iff there exists a path beginning at the given state, such that:

- — the OBE Property that is the right operand holds at the current state or at some future state; and
- — the OBE Property that is the left operand holds at all states, up to but not necessarily including, the state in which the OBE Property that is the right operand holds.

6.2.2.3 Logical OBE properties

6.2.2.3.1 OBE implication

The *OBE implication* operator (`->`), shown in Box 73, is used to specify logical implication.

```
OBE_Property ::=
        OBE_Property   -> OBE_Property
```

Box 73—OBE implication operator

The right operand of the OBE implication operator is an OBE Property that is specified to hold if the OBE Property that is the left operand holds.

Restrictions

None.

Informal semantics

An OBE implication property holds in a given state iff:

— the OBE property that is the left operand does not hold at the given state or
— the OBE property that is the right operand does hold at the given state.

6.2.2.3.2 OBE iff

The *OBE iff* operator (`<->`), shown in Box 74, is used to specify the *iff* (if and only if) relation between two properties.

```
OBE_Property ::=
        OBE_Property <-> OBE_Property
```

Box 74—OBE iff operator

The two operands of the OBE iff operator are OBE Properties. The OBE iff operator specifies that either both operands hold or neither operand holds.

Restrictions

None.

Informal semantics

An OBE iff property holds in a given state iff:

— both OBE Properties that are operands hold at the given state or
— neither of the OBE Properties that are operands hold at the given state.

6.2.2.3.3 OBE and

The *OBE and* operator, shown in Box 75, is used to specify logical and.

```
OBE_Property ::=
        OBE_Property  AND_OP OBE_Property
```

Box 75—OBE and operator

The operands of the OBE and operator are two OBE Properties that are both specified to hold.

Restrictions

None.

Informal semantics

An OBE and property holds in a given state iff the OBE Properties that are the operands both hold at the given state.

6.2.2.3.4 OBE or

The *OBE or* operator, shown in Box 76, is used to specify logical or.

```
OBE_Property ::=
        OBE_Property  OR_OP OBE_Property
```

Box 76—OBE or operator

The operands of the OBE or operator are two OBE Properties, at least one of which is specified to hold.

Restrictions

None.

Informal semantics

A OBE or property holds in a given state iff at least one of the OBE Properties that are the operands holds at the given state.

6.2.2.3.5 OBE not

The *OBE not* operator, shown in Box 77, is used to specify logical negation.

```
OBE_Property ::=
        NOT_OP OBE_Property
```

Box 77—OBE not operator

The operand of the OBE not operator is an OBE Property that is specified to not hold.

Restrictions

None.

Informal semantics

An OBE not property holds in a given state iff the OBE Property that is the operand does not hold at the given state.

6.2.3 Replicated properties

Replicated properties are specified using the operator `forall`, as shown in Box 78. The first operand of the replicated property is a `Replicator` and the second operand is a parameterized property.

```
Property ::=
        Replicator Property
Replicator ::=
        forall PSL_Identifier [ Index_Range ] in Value_Set :
Index_Range ::=
        LEFT_SYM finite_Range RIGHT_SYM
Flavor Macro LEFT_SYM =
        Verilog: [ / VHDL: ( / GDL: (
Flavor Macro RIGHT_SYM =
        Verilog: ] / VHDL: ) / GDL: )
Value_Set ::=
          { Value_Range { , Value_Range } }
        | boolean
Value_Range ::=
          Value
        | finite_Range
Range ::=
        Low_Bound RANGE_SYM High_Bound
```

Box 78—Replicating properties

The PSL Identifier in the replicator is the name of the parameter in the parameterized property. This parameter can be an array. The Value Set defines the set of values over which replication occurs.

1) If the parameter is not an array, then the property is replicated once for each value in the set of values, with that value substituted for the parameter. The total number of replications is equal to the size of the set of values.

2) If the parameter is an array of size *N*, then the property is replicated once for each possible combination of *N* (not necessarily distinct) values from the set of values, with those values substituted for the *N* elements of the array parameter. If the set of values has size *K*, then the total number of replications is equal to *K^N*.

The set of values can be specified in three different ways

— The keyword **boolean** specifies the set of values {*True*, *False*}.
— A `Value Range` specifies the set of all values within the given range.
— The comma (**,**) between *Value Ranges* indicates the union of the obtained sets.

Restrictions

If the parameter name has an associated Index Range, the Index Range shall be specified as a finite Range, each bound of the Range shall be statically computable, and the left bound of the Range shall be less than or equal to the right bound of the Range.

If a Value is used to specify a Value Range, the Value shall be statically computable.

If a Range is used to specify a Value Range, the Range shall be a finite Range, each bound of the Range shall be statically computable, and the left bound of the Range shall be less than or equal to the right bound of the Range.

The parameter name shall be used in one or more expressions in the Property, or as an actual parameter in the instantiation of a parameterized Property, so that each of the replicated instances of the Property corresponds to a unique value of the parameter name.

An implementation may impose restrictions on the use of a replication parameter name defined by a Replicator. However, an implementation shall support at least comparison (equality, inequality) between the parameter name and an expression, and use of the parameter name as an index or repetition count.

A replicator may appear in the declaration of a named property, provided that instantiations of the named property do not violate the above restriction. This means that the replicator must be at the top level of the named property declaration, and the named property's instantiations must not appear inside non-replicated properties.

Note—The parameter defined by a replicator is considered a static variable, and therefore the parameter name can be used in a static expression, such as that required by a repetition count.

Informal semantics

— A `forall i` in `boolean: f(i)` property is replicated to define two instances of the property `f(i):`

```
f(true)
f(false)
```

— A `forall i in {j:k} : f(i)` property is replicated to define *k-j*+1 instances of the property `f(i):`

```
f(j)
f(j+1)
f(j+2)
...
f(k)
```

— A `forall i in {j,l} :  f(i)` property is replicated to define two instances of the property `f(i)`:

```
f(j)
f(l)
```

— A `forall i[0:1] in boolean : f(i)` property is replicated to define four instances of the property `f(i)`:

```
f({false,false})
f({false,true})
f({true,false})
f({true,true})
```

— A `forall i[0:2] in {4,5} : f(i)` property is replicated to define eight instances of the property `f(i)`:

```
f({4,4,4})
f({4,4,5})
f({4,5,4})
f({4,5,5})
f({5,4,4})
f({5,4,5})
f({5,5,4})
f({5,5,5})
```

Examples

Legal:

```
forall i[0:3] in boolean:
    request && (data_in == i) -> next(data_out == i)

forall i in boolean:
  forall j in {0:7}:
      forall k in {0:3}:
         f(i,j,k)
```

Illegal:

```
always (request ->
    forall i in boolean: next_e[1:10](response[i]))

forall j in {0:7}:
    forall k in {0:j}:
       f(j,k)
```

6.2.4 Named properties

A given property may be applicable in more than one part of the design. In such a case, it is convenient to be able to define the property once and refer to the single definition wherever the property applies. Declaration and instantiation of *named properties* provide this capability.

6.2.4.1 Property declaration

A *property declaration*, shown in Box 79, defines a property and gives it a name. A property declaration can also specify a list of formal parameters that can be referenced within the property.

```
PSL_Declaration ::=
        Property_Declaration
Property_Declaration ::=
        property PSL_Identifier [ ( Formal_Parameter_List ) ] DEF_SYM Property ;
Formal_Parameter_List ::=
        Formal_Parameter { ; Formal_Parameter }
Formal_Parameter ::=
        Param_Type PSL_Identifier { , PSL_Identifier }
Param_Type ::=
        const | boolean | property | sequence
```

Box 79—Property declaration

Restrictions

The name of a declared endpoint shall not be the same as the name of any other PSL declaration in the same verification unit.

Informal Semantics

The PSL identifier following the keyword `property` in the property declaration is the name of the property. The PSL identifiers given in the formal parameter list are the names of the formal parameters of the named property.

Example

```
property ResultAfterN (boolean start; property result; const n; boolean stop) =
    always ((start -> next[n] (result)) @ (posedge clk) abort stop);
```

This property could also be declared as follows:

```
property ResultAfterN (boolean start, stop; property result; const n) =
    always ((start -> next[n] (result)) @ (posedge clk) abort stop);
```

The two declarations have slightly different interfaces (i.e., different formal parameter orders), but they both declare the same property `ResultAfterN`. This property describes behavior in which a specified result (a property) occurs n cycles after an enabling condition (parameter start) occurs, with cycles defined by rising edges of signal clk, unless an (asynchronous) abort condition (parameter stop) occurs.

NOTE—There is no requirement to use formal parameters in a property declaration. A declared property may refer directly to signals in the design as well as to formal parameters.

6.2.4.2 Property instantiation

A *property instantiation*, shown in Box 80, creates an instance of a named property and provides actual parameters for formal parameters (if any) of the named property.

```
FL_Property ::=
        property_Name [ ( Actual_Parameter_List ) ]
Actual_Parameter_List ::=
        Actual_Parameter { , Actual_Parameter }
Actual_Parameter ::=
        Number | Boolean | Property | Sequence
```

Box 80—Property instantiation

Restrictions

For each formal parameter of the named property, the property instantiation shall provide a corresponding actual parameter. For a `const` formal parameter, the actual parameter shall be a statically evaluable integer expression. For a `boolean` formal parameter, the actual parameter shall be a Boolean expression. For a `property` formal parameter, the actual parameter shall be an FL Property. For a `sequence` formal parameter, the actual parameter shall be a Sequence.

Informal semantics

An instance of a named property holds at a given evaluation cycle if and only if the named property, modified by replacing each formal parameter in the property declaration with the corresponding actual parameter in the property instantiation, holds in that evaluation cycle.

Example

Given the first declaration for the property `ResultAfterN` in 6.2.4.1,

```
   ResultAfterN (write_req, eventually! ack, 3, cancel)
   ResultAfterN (read_req, eventually! (ack | retry), 5,
       (cancel | write_req))
```

is equivalent to

```
   always ((write_req -> next[3] (eventually! ack)) @ (posedge clk) abort
       cancel)
   always ((read_req -> next[5] (eventually! (ack | retry))) @ (posedge clk)
       abort (cancel | write_req))
```

7. Verification layer

The verification layer provides *directives* that tell a verification tool what to do with specified sequences and properties. The verification layer also provides constructs that group related directives and other PSL statements.

7.1 Verification directives

Verification directives give directions to verification tools.

```
PSL_Directive ::=
   [ Label : ] Verification_Directive

Verification_Directive ::=
   Assert_Directive
 | Assume_Directive
 | Assume_Guarantee_Directive
 | Restrict_Directive
 | Restrict_Guarantee_Directive
 | Cover_Directive
 | Fairness_Statement
```

Box 81—Verification Directives

A verification directive may be preceded by a label. If present, the label must not be the same as any other label in the same verification unit.

Labels enable construction of a unique name for any instance of that directive. Such unique names can be used by a tool for selective control and reporting of results.

```
Label ::=
   PSL_Identifier
```

Box 82—Labels

NOTE—Labels cannot be referenced from other PSL constructs. They are provided only to enable unique identification of PSL directives within tool graphical interfaces and textual reports.

7.1.1 assert

The verification directive `assert`, shown in Box 83, instructs the verification tool to verify that a property holds.

```
Assert_Directive ::=
   assert Property [ report String ] ;
```

Box 83—Assert statement

An assert directive may optionally include a character string containing a message to report when the property fails to hold.

Example

The directive

```
   assert always (ack -> next (!ack until req))
           report “A second ack occurred before the next req”;
```

instructs the verification tool to verify that the property

```
       always (ack -> next (!ack until req))
```

holds in the design. If the verification tool discovers a situation in which this property does not hold, it should display the message:

```
       A second ack occurred before the next req
```

7.1.2 assume

The verification directive `assume`, shown in Box 84, instructs the verification tool to constrain the verification (e.g., the behavior of the input signals) so that a property holds.

```
Assume_Directive ::=
   assume Property ;
```

Box 84—Assume statement

Restrictions

The Property that is the operand of an `assume` directive must be an FL Property or replicated FL Property.

Example

The directive

```
   assume always (ack -> next !ack);
```

instructs the verification tool to constrain the verification (e.g., the behavior of the input signals) so that the property

```
   always (ack -> next !ack)
```

holds in the design.

Assumptions are often used to specify the operating conditions of a design property by constraining the behavior of the design inputs. In other words, an asserted property is required to hold only along those paths that obey the assumption.

NOTE—Verification tools are not obligated to verify the assumed property.

7.1.3 assume_guarantee

The `assume_guarantee` directive, shown in Box 85, instructs the verification tool to constrain the verification (e.g., the behavior of the input signals) so that a property holds and also to verify that the assumed property holds.

Assume_Guarantee_Directive ::=
  assume_guarantee Property [**report** String] **;**

Box 85—Assume_guarantee statement

An assume_guarantee directive may optionally include a character string containing a message to report when the property fails to hold.

Restrictions

The Property that is the operand of an `assume_guarantee` directive must be an FL Property or replicated FL Property.

Example

The directive

```
assume_guarantee always (ack -> next !ack);
```

instructs the tool to assume that whenever signal `ack` is asserted, it is not asserted at the next cycle, while also verifying that the property holds. To illustrate how this verification directive is used, imagine two design blocks, `A` and `B`, and the signal `ack` as an output from block `B` and an input to block `A`. The property

```
assume_guarantee always (ack -> next !ack);
```

can be assumed to verify some other properties related to block `A`. However, verification tools shall also indicate the proof obligation of this property when block `B` is present. How this information is used is tool-dependent.

NOTE—The optional character string has no effect when an assume_guarantee directive is being used only to indicate an assumption. The character string can be provided so that it can be reported when the property fails to verify in another context.

7.1.4 restrict

The verification directive restrict, shown in Box 86, is a way to constrain the design inputs using sequences.

Restrict_Directive ::=
  restrict Sequence **;**

Box 86—Restrict statement

A restrict directive can be used to initialize the design to get to a specific state before checking assertions.

NOTE—Verification tools are not obligated to verify that the restricted sequence holds.

Example

The directive

```
restrict {!rst;rst[*3];!rst[*]};
```

is a constraint that every execution trace begins with one cycle of rst low, followed by three cycles of rst high, followed by rst being low forever.

7.1.5 restrict_guarantee

The directive restrict_guarantee, shown in Box 87, instructs the verification tool to constrain the design inputs so that a sequence holds and also to verify that the restrict sequence holds.

Restrict_Directive ::=
  restrict_guarantee Sequence [**report** String] **;**

Box 87—Restrict_guarantee statement

A restrict_guarantee directive may optionally include a character string containing a message to report when the sequence fails to hold.

Example

The directive

```
restrict_guarantee {!rst;rst[*3];!rst[*]};
```

is a constraint that every execution trace begins with one cycle of `rst` low, followed by three cycles of `rst` high, followed by `rst` being low forever, while also verifying that the constraint holds. How this information is used is tool-dependent.

NOTE—The optional character string has no effect when a restrict_guarantee directive is being used only to indicate a restriction. The character string can be provided so that it can be reported when an attempt to verify that the sequence holds.

7.1.6 cover

The verification directive `cover`, shown in Box 88, directs the verification tool to check if a certain path was covered by the verification space based on a simulation test suite or a set of given constraints.

Cover_Directive ::=
  cover Sequence [**report** String] **;**

Box 88—Cover statement

A cover directive may optionally include a character string containing a message to report when the specified sequence occurs.

Example

The directive

```
cover {start_trans;!end_trans[*];start_trans & end_trans}
        report “Transactions overlapping by one cycle covered” ;
```

instructs the verification tool to check if there is at least one case in which a transaction starts and then another one starts the same cycle that the previous one completed.

Note that `cover {r}` is semantically equivalent to `cover {[*];r}`. That is, there is an implicit [*] starting the sequence.

7.1.7 fairness and strong fairness

The directives `fairness` and `strong fairness`, shown in Box 89, are special kinds of assumptions that correspond to liveness properties.

Fairness_Statement ::=
  fairness Boolean **;**
  | **strong fairness** Boolean **,** Boolean **;**

Box 89—Fairness statement

If the fairness constraint includes the keyword `strong`, then it is a *strong fairness constraint*; otherwise it is a *simple fairness constraint.*

Fairness constraints can be used to filter out certain behaviors. For example, they can be used to filter out a repeated occurrence of an event that blocks another event forever. Fairness constraints guide the verification tool to verify the property only over fair paths. A path is *fair* if every fairness constraint holds along the path. A simple fairness constraint holds along a path if the given Boolean expression occurs infinitely many times along the path. A strong fairness constraint holds along the path if a given Boolean expression does not occur infinitely many times along the path or if another given Boolean expression occurs infinitely many times along the path.

Examples

The directive

```
fairness p;
```

instructs the verification tool to verify the formula only over paths in which the Boolean expression p occurs infinitely often. Semantically it is equivalent to the assumption

```
assume G F p;
```

The directive

```
strong fairness p,q;
```

instructs the verification tool to verify the formula only over paths in which either the Boolean expression p does not occur infinitely often or the Boolean expression q occurs infinitely often. Semantically it is equivalent to the assumption

```
assume (G F p) -> (G F q);
```

7.2 Verification units

A *verification unit*, shown in Box 90, is used to group verification directives and other PSL statements.

```
Verification_Unit ::=
    Vunit_Type PSL_Identifier [ ( Hierarchical_HDL_Name ) ] {
        { Inherit_Spec }
        { VUnit_Item }
    }
Vunit_Type ::=
    vunit | vprop | vmode
Hierarchical_HDL_Name ::=
    HDL_MOD_NAME { Path_Separator  instance_Name }
HDL_MOD_NAME =
     SystemVerilog: module_Name
   / Verilog: module_Name
   / VHDL: entity_aspect
   / GDL: module_Name
Path_Separator ::=
   . | /
Name ::=
    HDL_or_PSL_Identifier
Inherit_Spec ::=
   inherit vunit_Name { , vunit_Name } ;
VUnit_Item ::=
    HDL_DECL
  | HDL_STMT
  | PSL_Declaration
  | PSL_Directive
```

Box 90—Verification unit

The PSL Identifier following the keyword `vunit` is the name by which this verification unit is known to the verification tools.

If the Hierarchical HDL Name is present, then the verification unit is explicitly bound to the specified design module or module instance. If the Hierarchical HDL Name is not present, then the verification unit is not explicitly bound. See 7.2.1 for a discussion of binding.

An Inherit Spec indicates another verification unit from which this verification unit inherits contents. See 7.2.2 for a discussion of inheritance.

A VUnit Item can be any of the following:

a) Any modeling layer statement or declaration.
b) A property, endpoint, sequence, or default clock declaration.
c) Any verification directive.

The Vunit Type specifies the type of the Verification Unit. Verification unit types `vprop` and `vmode` enable separate definition of assertions to verify and constraints (i.e., assumptions or restrictions) to be considered in attempting to verify those assertions. Various `vprop` verification units can be created containing different sets of assertions to verify and various `vmode` verification units containing different sets of constraints can be created to represent the different conditions under which verification should take place. By combining one or more `vprop` verification units with one or more `vmode` verification units, the user can easily compose different verification tasks.

Verification unit type `vunit` enables a combined approach in which both assertions to verify and applicable constraints, if any, can be defined together. All three types of verification units can be used together in a single verification run.

The default verification unit (i.e., one named `default`) can be used to define constraints that are common to all verification environments, or defaults that can be overridden in other verification units. For example, the default verification unit might include a default clock declaration or a sequence declaration for the most common reset sequence.

Restrictions

A Verification Unit of type vmode shall not contain an assert directive.

A Verification Unit of type vprop shall not contain a directive that is not an assert directive.

A Verification Unit of type vprop shall not inherit a Verification Unit of type vunit or vmode.

A Verification Unit of type vmode shall not inherit a Verification Unit of type vunit or vprop.

A default Verification Unit, if it exists, shall be of type vmode. The default vmode shall not inherit other verification units of any type.

7.2.1 Verification unit binding

The connection between signals referred to in a verification unit and signals of the design under verification is by name, relative to the module or module instance to which the verification unit is bound.

If the verification unit is explicitly bound to an instance, then that instance is the context used to interpret HDL names and operator symbols, as defined in section Section 5.2.1, HDL expressions.

If the verification unit is explicitly bound to a module, then this is equivalent to duplicating the contents of the verification unit and binding each duplication to one instance.

If the verification unit is not explicitly bound, then a verification tool may allow the user to specify the binding of the verification unit separate from the verification unit. A verification unit that is not explicitly bound can also be used to group together commonly used PSL declarations so they can be inherited for use in other PSL verification units.

Examples

```
vunit ex1a(top_block.i1.i2) {
  A1: assert never (ena && enb);
}
```

`vunit ex1a` is bound to instance `top_block.i1.i2`. This causes assertion A1 to apply to signals ena and enb in top_block.i1.i2.

As a second example, consider:

```
vunit ex2a(mod1) {
  A2: assert never (ena && enb);
}
```

The verification unit is bound to module `mod1`. If this module is instantiated twice in the design, once as `top_block.i1.i2` and once as `top_block.i1.i3`, then `vunit ex2a` is equivalent to the following pair of vunits:

```
vunit ex2b(top_block.i1.i2) {
      A2: assert never(ena && enb);
    }
vunit ex2c(top_block.i1.i3) {
      A2: assert never(ena && enb);
    }
```

As a third example, consider:

```
vunit ex3 {
  A3: assert never (ena && enb);
}
```

This verification unit is not explicitly bound, so there is no context in which to interpret references to '`ena`' and '`enb`'. In this case, the verification tool may determine the binding.

Finally, consider:

```
vunit ex4 {
  property mutex (boolean b1, b2) = never (b1 && b2);
}
```

This verification unit is not explicitly bound, however it contains no HDL expressions that require interpretation, so no binding is necessary. This illustrates use of an unbound verification unit for commonly used PSL declarations that can be inherited into other verification units.

7.2.2 Verification unit inheritance

One verification unit may inherit one or more other verification units, each of which may inherit other verification units, and so on. Inheritance is transitive.

For a verification unit that inherits one or more other units, its inherited context is the set of verification units in the transitive closure with respect to inheritance. A verification unit must not be contained in its own inherited context.

Inheritance has two effects:

a) As a consequence of the rules for determining the meaning of names (in Section 5.2.1), any PSL declarations in a given verification unit's inherited context can be referenced in that verification unit.

b) When a given verification unit is considered by a verification tool, the contents of that verification unit and its inherited context are taken together to define the environment in which verification is to take place, and the set of directives to consider during verification.

Examples

Assume there are two blocks, A and B, which are mutually dependent—the outputs of A (`Aout1`, `Aout2`) are inputs of B (`Bin1`, `Bin2`), and vice versa. The following verification units might describe the interactions between the two blocks:

```
vmode Common {
    property mutex (boolean b1, b2) = never b1 && b2 ;
    property one_hot (boolean b1, b2) =
                            always ((b1 &&! b2) || (b2 && !b1));
}
```

Verification unit Common is not explicitly bound. It contains commonly used property definitions. It is declared as a 'vmode' so it can be inherited by other vmode units.

```
vmode Amode (blockA) {
    inherit Common;
    assume mutex(Aout1, Aout2);
}
vmode Bmode (blockB) {
    inherit Common;
    assume one_hot(Bout1, Bout2);
}
```

Verification units `Amode` and `Bmode` contain assumptions about these blocks to be made when verifying properties in other blocks. They are both explicitly bound, so that the HDL name references they contain have meaning. They both inherit the Common vmode in order to make use of the property declarations it contains.

```
vunit Aprops (blockA) {
    inherit Common, Bmode;
    assert mutex(Aout1, Aout2);
}
vunit Bprops (blockB) {
    inherit Common, Amode;
    assert one_hot(Bout1, Bout2);
}
```

Verification units `Aprops` and `Bprops` contain assertions to verify about the respective blocks. They are both explicitly bound, so that the HDL name references they contain have meaning. They both inherit the Common vmode in order to make use of its property declarations. The vunit `Aprops` inherits vmode `Bmode` so that assumptions about the outputs of B can be considered when verifying the behavior of block A. The vunit `Bprops` inherits vmode `Amode` so that assumptions about the outputs of A can be considered when verifying the behavior of block B.

7.2.3 Verification unit scoping rules

If a verification unit that is bound to a given HDL instance contains a modeling layer declaration, then that verification unit's declaration takes precedence over any other declaration of the same name, either in a verification unit bound to the same HDL instance in its inherited context, or in the HDL instance itself. This allows a verification unit to redeclare and/or give new behavior to a signal in the design under verification.

Example

Consider the following verification unit:

```
vunit ex5a(top_block.i1) {
  wire temp;
  assign temp = ack1 || ack2;
  A5: assert always (reqa -> next temp);
}
```

The vunit `ex5a` declares wire temp and assigns it a value. This could be just an auxiliary statement to facilitate specification of property `A5`. However, if instance `top_block.i1` also contains a declaration of a signal named '`temp`', then the declaration in `ex5a` would override the declaration in the design, and the assignment to '`temp`' in vunit `ex5a` would override the driving logic for signal '`temp`' in the design.

Now consider the following verification unit:

```
vunit ex5b(top_block.i1) {
  inherit ex5a;
  wire temp;
  assign temp = ack1 || ack2 || ack3;
  A6: assert always (reqb -> next temp);
}
```

Verification unit `ex5b` inherits `ex5a`. Both verification units are bound to the same instance and both declare wires named temp. The declaration of temp in the inheriting verification unit takes precedence, so the declaration of (and assignment to) temp in `ex5b` takes precedence when verifying `ex5b`, and the declaration of (and assignment to) temp in both the design and vunit `ex5a` are ignored.

8. Modeling layer

The modeling layer provides a means to model behavior of design inputs (for tools such as formal verification tools in which the behavior is not otherwise specified), and to declare and give behavior to auxiliary signals and variables. The modeling layer comes in four flavors, corresponding to SystemVerilog, Verilog, VHDL, and GDL.

The SystemVerilog flavor of the modeling layer will consist of the synthesizable subset of SystemVerilog, which is not yet defined. The SystemVerilog flavor of the modeling layer extends SystemVerilog to include integer range declarations, as defined in section 8.1.

The Verilog flavor of the modeling layer consists of the synthesizable subset of Verilog, defined by IEEE standard 1364.1-2002, Standard for Verilog Register Transfer Level Synthesis. The Verilog flavor of the modeling layer extends Verilog to include integer range declarations, as defined in section 8.1, and struct declarations, as defined in section 8.2.

The VHDL flavor of the modeling layer consists of the synthesizable subset of VHDL, defined by IEEE Std 1076.6-1999, IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis.

The GDL flavor of the modeling layer consists of all of GDL.

In each flavor of the modeling layer, at any place where an HDL expression may appear, the modeling layer is extended to allow any form of HDL or PSL expression, as defined in section 5, Boolean Layer. Thus HDL expressions, PSL expressions, built-in functions, endpoints, and union expressions may all be used as expressions within the modeling layer.

Each flavor of the modeling layer supports the comment constructs of the corresponding hardware description language.

8.1 Integer ranges

The SystemVerilog and Verilog flavors of the modeling layer are extended to include declaration of a finite integer type, shown in Box 91, where the range of values that the variable can take on is indicated by the declaration.

```
Finite_Integer_Type_Declaration ::=
      integer Integer_Range list_of_variable_identifiers ;
Integer_Range ::=
      ( constant_expression : constant_expression )
```

Box 91—integer range declaration

The nonterminals list_of_variable_identifiers and constant_expression are defined in the syntax for IEEE 1364-2001 Verilog and in the syntax for Accellera SystemVerilog 3.1a..

Example

```
integer (1:5) a, b[1:20];
```

This declares an integer variable `a`, which can take on values between `1` and `5`, inclusive, and an integer array `b`, each of whose twenty entries can take on values between `1` and `5`, inclusive.

8.2 Structures

The Verilog flavor of the modeling layer also extends the Verilog data types to allow declaration of C-like structures, as shown in Box 92.

```
Structure_Type_Declaration ::=
    struct { Declaration_List } list_of_variable_identifiers ;
Declaration_List ::=
    HDL_Variable_or_Net_Declaration { HDL_Variable_or_Net_Declaration }
HDL_Variable_or_Net_Declaration ::=
      net_declaration
    | reg_declaration
    | integer_declaration
```

Box 92—Structure declaration

The nonterminals list_of_variable_identifiers, net_declaration, reg_declaration, and integer_declaration are defined in the syntax for IEEE 1364-2001 Verilog.

Example

```
struct {
  wire w1, w2;
  reg r;
  integer(0..7) i;
} s1, s2;
```

which declares two structures, `s1` and `s2`, each with four fields, `w1`, `w2`, `r`, and `i`. Structure fields are accessed as `s1.w1`, `s1.w2`, etc.

Appendix A

(normative)

Syntax rule summary

The appendix summarizes the syntax.

A.1 Meta-syntax

The formal syntax described in this standard uses the following extended Backus-Naur Form (BNF).

a) The initial character of each word in a nonterminal is capitalized. For example:

    PSL_Statement

   A nonterminal can be either a single word or multiple words separated by underscores. When a multiple-word nonterminal containing underscores is referenced within the text (e.g., in a statement that describes the semantics of the corresponding syntax), the underscores are replaced with spaces.

b) Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. For example:

    vunit (;

c) The **::=** operator separates the two parts of a BNF syntax definition. The syntax category appears to the left of this operator and the syntax description appears to the right of the operator. For example, item d) shows three options for a *Vunit_Type*.

d) A vertical bar separates alternative items (use one only) unless it appears in boldface, in which case it stands for itself. For example:

    Vunit_Type ::= **vunit** | **vprop** | **vmode**

e) Square brackets enclose optional items unless it appears in boldface, in which case it stands for itself. For example:

    Sequence_Declaration ::=
      sequence Name [**(** Formal_Parameter_List **)**] DEF_SYM Sequence **;**

   indicates *Formal_Parameter_List* is an optional syntax item for *Sequence_Declaration*, whereas

    | Sequence **[*** [Range] **]**

   indicates that (the outer) square brackets are part of the syntax, while *Range* is optional.

f) Braces enclose a repeated item unless it appears in boldface, in which case it stands for itself. A repeated 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:

    Formal_Parameter_List ::= Formal_Parameter { **;** Formal_Parameter }
    Formal_Parameter_List ::= Formal_Parameter | Formal_Parameter_List **;** Formal_Parameter

g) A comment in a production is preceded by a colon (:) unless it appears in boldface, in which case it stands for itself.

h) 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, *vunit_*Name is equivalent to Name.

i) Flavor macros, containing embedded underscores, are shown in uppercase. These reflect the various HDLs that can be used within the PSL syntax and show the definition for each HDL. The general format is the term `Flavor Macro`, then the actual *macro name*, followed by the `=` operator, and, finally, the definition for each of the HDLs. For example:

   Flavor Macro RANGE_SYM =
      SystemVerilog: **:** / Verilog: **:** / VHDL: **to** / GDL: **/ ..**

   shows the *range symbol* macro (RANGE_SYM). See 4.3.2 for further details about *flavor macros*.

The main text uses *italicized* type when a term is being defined, and `monospace` font for examples and references to constants such as `0`, `1`, or `X` values.

A.2 Tokens

PSL syntax is defined in terms of primitive *tokens*, which are character sequences that act as distinct symbols in the language.

Each PSL keyword is a single token. Some keywords end in one or two non-alphabetic characters ('!' or '_' or both). Those characters are part of the keyword, not separate tokens.

Each of the following character sequences is also a token:

[	**]**	**(**	**)**	**{**	**}**
,	**;**	**:**	**..**	**=**	**:=**
*****	**+**	**|->**	**|=>**	**<->**	**->**
[*	**[+]**	**[->**	**[=**		
&&	**&**	**||**	**|**	**!**	
$	**@**	**.**	**/**		

Finally, for a given flavor, the tokens of the corresponding HDL are tokens of PSL.

A.3 HDL Dependencies

PSL depends upon the syntax and semantics of an underlying hardware description language. In particular, PSL syntax includes productions that refer to nonterminals in SystemVerilog, Verilog, VHDL, or GDL. PSL syntax also includes Flavor Macros that cause each flavor of PSL to match that of the underlying HDL for that flavor.

For SystemVerilog, the PSL syntax refers to the following nonterminals in the Accellera SystemVerilog version 3.1a syntax:

— module_or_generate_item_declaration
— module_or_generate_item
— list_of_variable_identifiers
— identifier
— expression
— constant_expression

For Verilog, the PSL syntax refers to the following nonterminals in the IEEE 1364-2001 Verilog syntax:

— module_or_generate_item_declaration
— module_or_generate_item
— list_of_variable_identifiers
— identifier
— expression
— constant_expression
— net_declaration
— reg_declaration
— integer_declaration

For VHDL, the PSL syntax refers to the following nonterminals in the IEEE 1076-1993 VHDL syntax:

— block_declarative_item
— concurrent_statement
— design_unit
— identifer
— expression
— entity_aspect

For GDL, the PSL syntax refers to the following nonterminals in the GDL syntax:

— module_item_declaration
— module_item
— module_declaration
— identifer
— expression

A.3.1 Verilog Extensions

For the Verilog flavor, PSL extends the forms of declaration that can be used in the modeling layer by defining two additional forms of type declaration. PSL also adds an additional form of expression for both Verilog and VHDL flavors.

Extended_Verilog_Declaration ::=
  *Verilog*_module_or_generate_item_declaration
 | Extended_Verilog_Type_Declaration

Extended_Verilog_Type_Declaration ::=
  integer Integer_Range list_of_variable_identifiers **;**
 | **struct {** Declaration_List **}** list_of_variable_identifiers **;**

Integer_Range ::=
 (constant_expression **:** constant_expression **)**

```
Declaration_List ::=
        HDL_Variable_or_Net_Declaration { HDL_Variable_or_Net_Declaration }

HDL_Variable_or_Net_Declaration ::=
          net_declaration
        | reg_declaration
        | integer_declaration
```

A.3.2 Flavor macros

Flavor Macro DEF_SYM =
 SystemVerilog: **=** / Verilog: **=** / VHDL: **is** / GDL: **:=**

Flavor Macro RANGE_SYM =
 SystemVerilog: **:** / Verilog: **:** / VHDL: **to** / GDL: **..**

Flavor Macro AND_OP =
 SystemVerilog: **&&** / Verilog: **&&** / VHDL: **and** / GDL: **&**

Flavor Macro OR_OP =
 SystemVerilog: **||** / Verilog: **||** / VHDL: **or** / GDL: **|**

Flavor Macro NOT_OP =
 SystemVerilog: **!** / Verilog: **!** / VHDL: **not** / GDL: **!**

Flavor Macro MIN_VAL =
 SystemVerilog: **0** / Verilog: **0** / VHDL: **0** / GDL: *null*

Flavor Macro MAX_VAL =
 SystemVerilog: **$** / Verilog: **inf** / VHDL: **inf** / GDL: *null*

Flavor Macro HDL_EXPR =
 SystemVerilog: SystemVerilog_Expression
 / Verilog: Verilog_Expression
 / VHDL: Extended_VHDL_Expression
 / GDL: GDL_Expression

Flavor Macro HDL_CLK_EXPR =
 SystemVerilog: SystemVerilog_Event_Expression
 / Verilog: Verilog_Event_Expression
 / VHDL: VHDL_Expression
 / GDL: GDL_Expression

Flavor Macro HDL_UNIT =
 SystemVerilog: SystemVerilog_module_declaration
 / Verilog: *Verilog*_module_declaration
 / VHDL: *VHDL*_design_unit
 / GDL: *GDL*_module_declaration

Flavor Macro HDL_MOD_NAME =
 SystemVerilog: *module*_Name
 / Verilog: *module*_Name
 / VHDL: entity_aspect
 / GDL: *module*_Name

Flavor Macro HDL_DECL =
    SystemVerilog: SystemVerilog_module_or_generate_item_declaration
    / Verilog: Extended_Verilog_Declaration
    / VHDL: *VHDL*_block_declarative_item
    / GDL: *GDL*_module_item_declaration

Flavor Macro HDL_STMT =
    SystemVerilog: SystemVerilog_module_or_generate_item
    / Verilog: *Verilog*_module_or_generate_item
    / VHDL: *VHDL*_concurrent_statement
    / GDL: *GDL*_module_item

Flavor Macro HDL_RANGE =
    VHDL: range_attribute_name

Flavor Macro LEFT_SYM =
    SystemVerilog: **[** / Verilog: **[** / VHDL: **(** / GDL: **(**

Flavor Macro RIGHT_SYM =
    SystemVerilog: **]** / Verilog: **]** / VHDL: **)** / GDL: **)**

A.4 Syntax productions

The rest of this section defines the PSL syntax.

A.4.1 Verification units

PSL_Specification ::=
    { Verification_Item }

Verification_Item ::=
    HDL_UNIT | Verification_Unit

Verification_Unit ::=
    Vunit_Type *PSL*_Identifier [**(** Hierarchical_HDL_Name **)**] **{**
        { Inherit_Spec }
        { VUnit_Item }
    }

Vunit_Type ::=
    vunit | **vprop** | **vmode**

Name ::=
    HDL_or_PSL_Identifier

Hierarchical_HDL_Name ::=
    HDL_MOD_NAME { Path_Separator *instance*_Name }

Path_Separator ::=
    . | **/**

Inherit_Spec ::=
  inherit *vunit*_Name { **,** *vunit*_Name } **;**

VUnit_Item ::=
    HDL_DECL
  | HDL_STMT
  | PSL_Declaration (see A.4.2)
  | PSL_Directive (see A.4.3)

A.4.2 PSL declarations

PSL_Declaration ::=
    Property_Declaration
  | Sequence_Declaration
  | Endpoint_Declaration
  | Clock_Declaration

Property_Declaration ::=
  property *PSL*_Identifier [**(** Formal_Parameter_List **)**] DEF_SYM Property **;**

Formal_Parameter_List ::=
  Formal_Parameter { **;** Formal_Parameter }

Formal_Parameter ::=
  Param_Type *PSL*_Identifier { **,** *PSL*_Identifier }

Param_Type ::=
  const | **boolean** | **property** | **sequence**

Sequence_Declaration ::=
  sequence *PSL*_Identifier [**(** Formal_Parameter_List **)**] DEF_SYM Sequence **;** (see A.4.6)

Endpoint_Declaration ::=
  endpoint *PSL*_Identifier [**(** Formal_Parameter_List **)**] DEF_SYM Sequence **;** (see A.4.6)

Clock_Declaration ::=
  default clock DEF_SYM Clock_Expression **;** (see A.4.7)

Clock_Expression ::=
    *boolean*_Name
  | *boolean*_Built_In_Function_Call
  | Endpoint_Instance
  | **(** Boolean **)**
  | **(** HDL_CLK_EXPR **)**

Actual_Parameter_List ::=
  Actual_Parameter { **,** Actual_Parameter }

Actual_Parameter ::=
  Number | Boolean | Property | Sequence (see A.4.7) (see A.4.7) (see A.4.4) (see A.4.6)

A.4.3 PSL directives

PSL_Directive ::=
    [Label **:**] Verification_Directive

Label ::=
    *PSL*_Identifier

HDL_or_PSL_Identifier ::=
    *SystemVerilog*_Identifier
    | Verilog_Identifier
    | V*HDL*_Identifier
    | *GDL*_Identifier
    | *PSL*_Identifier

Verification_Directive ::=
    Assert_Directive
    | Assume_Directive
    | Assume_Guarantee_Directive
    | Restrict_Directive
    | Restrict_Guarantee_Directive
    | Cover_Directive
    | Fairness_Statement

Assert_Directive ::=
    assert Property **[report** String **] ;** (see A.4.4)

Assume_Directive ::=
    assume Property **;** (see A.4.4)

Assume_Guarantee_Directive ::=
    assume_guarantee Property **[report** String **] ;** (see A.4.4)

Restrict_Directive ::=
    restrict Sequence **;** (see A.4.6)

Restrict_Guarantee_Directive ::=
    restrict_guarantee Sequence [**report** String] **;** (see A.4.6)

Cover_Directive ::=
    cover Sequence **[report** String **] ;** (see A.4.6)

Fairness_Statement ::=
    fairness Boolean **;**
    | **strong fairness** Boolean **,** Boolean **;** (see A.4.7)

A.4.4 PSL properties

Property ::=
    Replicator Property
    | FL_Property
    | OBE_Property

```
Replicator ::=
        forall PSL_Identifier [ Index_Range ] in Value_Set :

Index_Range ::=
         LEFT_SYM finite_Range RIGHT_SYM
        | ( HDL_RANGE )

Value_Set ::=
         { Value_Range { , Value_Range } }
        | boolean

Value_Range ::=
         Value                                                    (see A.4.7)
        | FiniteRange

Value ::=
        Boolean
        | Number
                                                                  (see A.4.6)
FL_Property ::=
         Boolean                                                  (see A.4.7)
        | ( FL_Property )
        | Sequence [ ! ]
        | property_Name [ ( Actual_Parameter_List ) ]
        | FL_Property @ Clock_Expression
        | FL_Property abort Boolean
: Logical Operators :
        | NOT_OP FL_Property
        | FL_Property AND_OP FL_Property
        | FL_Property OR_OP FL_Property
        :
        | FL_Property -> FL_Property
        | FL_Property <-> FL_Property
: Primitive LTL Operators :
        | X FL_Property
        | X! FL_Property
        | F FL_Property
        | G FL_Property
        | [ FL_Property U FL_Property ]
        | [ FL_Property W FL_Property ]
: Simple Temporal Operators :
        | always FL_Property
        | never FL_Property
        | next FL_Property
        | next! FL_Property
        | eventually! FL_Property
        :
        | FL_Property until! FL_Property
        | FL_Property until FL_Property
        | FL_Property until!_ FL_Property
        | FL_Property until_ FL_Property
        :
        | FL_Property before! FL_Property
        | FL_Property before FL_Property
        | FL_Property before!_ FL_Property
        | FL_Property before_ FL_Property
```

: Extended Next (Event) Operators : (see A.4.7)
| **X [** Number **] (** FL_Property **)**
| **X! [** Number **] (** FL_Property **)**
| **next [** Number **] (** FL_Property **)**
| **next! [** Number **] (** FL_Property **)**
: (see A.4.6)
| **next_a [** *finite*_Range **] (** FL_Property **)**
| **next_a! [** *finite*_Range **] (** FL_Property **)**
| **next_e [** *finite*_Range **] (** FL_Property **)**
| **next_e! [** *finite*_Range **] (** FL_Property **)**
:
| **next_event! (** Boolean **) (** FL_Property **)**
| **next_event (** Boolean **) (** FL_Property **)**
| **next_event! (** Boolean **) [** *positive*_Number **] (** FL_Property **)**
| **next_event (** Boolean **) [** *positive*_Number **] (** FL_Property **)**
:
| **next_event_a! (** Boolean **) [** *finite_positive*_Range **] (** FL_Property **)**
| **next_event_a (** Boolean **) [** *finite_positive*_Range **] (** FL_Property **)**
| **next_event_e! (** Boolean **) [** *finite_positive*_Range **] (** FL_Property **)**
| **next_event_e (** Boolean **) [** *finite_positive*_Range **] (** FL_Property **)**
: Operators on SEREs : (see A.4.6)
| { Sequence } **(** FL_Property **)**
| Sequence **|->** FL_Property
| Sequence **|=>** FL_Property

A.4.5 Sequential Extended Regular Expressions (SEREs)

SERE ::=
Boolean
| Sequence
| Sequence_Instance
| SERE **;** SERE
| SERE **:** SERE
| Compound_SERE

Compound_SERE ::=
Repeated_SERE
| Braced_SERE
| Clocked_SERE
| Compound_SERE **|** Compound_SERE
| Compound_SERE **&** Compound_SERE
| Compound_SERE **&&** Compound_SERE
| Compound_SERE **within** Compound_SERE

A.4.6 Sequences

Sequence ::=
Sequence_Instance
| Repeated_SERE
| Braced_SERE
| Clocked_SERE

```
Repeated_SERE ::=
        Boolean [* [ Count ] ]
      | Sequence [* [ Count ] ]
      | [* [ Count ] ]
      | Boolean [+]
      | Sequence [+]
      | [+]
      | Boolean [= Count ]
      | Boolean [-> [ positive_Count ] ]

Braced_SERE ::=
        { SERE }

Sequence_Instance ::=
        sequence_Name [ ( Actual_Parameter_List ) ]

Clocked_SERE ::=
        Braced_SERE @ Clock_Expression

Count ::=
        Number
      | Range

Range ::=
        Low_Bound RANGE_SYM High_Bound

Low_Bound ::=
        Number
      | MIN_VAL

High_Bound ::=
        Number
      | MAX_VAL
```

A.4.7 Forms of expression

```
Any_Type  ::=
      HDL_or_PSL_Expression

Bit  ::=
      bit_HDL_or_PSL_Expression

Boolean  ::=
      boolean_HDL_or_PSL_Expression

BitVector  ::=
      bitvector_HDL_or_PSL_Expression

Number  ::=
      numeric_HDL_or_PSL_Expression
```

String ::=
    *string*_HDL_or_PSL_Expression

HDL_or_PSL_Expression ::=
    HDL_Expression
  | PSL_Expression
  | Built_In_Function_Call
  | Union_Expression
  | Endpoint_Instance

HDL_Expression ::=
    HDL_EXPR

PSL_Expression ::=
    Boolean **->** Boolean
  | Boolean **<->** Boolean

Built_In_Function_Call ::=
    prev (Any_Type [**,** Number] **)**
  | **next (** Any_Type **)**
  | **stable (** Any_Type **)**
  | **rose (** Bit **)**
  | **fell (** Bit **)**
  | **isunknown (** BitVector **)**
  | **countones (** BitVector **)**
  | **onehot (** BitVector **)**
  | **onehot0 (** BitVector **)**

Union_Expression ::=
  Any_Type **union** Any_Type

Endpoint_Instance ::=
  *endpoint*_Name [**(** Actual_Parameter_List **)**]

A.4.8 Optional branching extension

OBE_Property ::=
    Boolean
  | **(** OBE_Property **)**
  | *property*_Name [**(** Actual_Parameter_List **)**]

: Logical Operators :
  | NOT_OP OBE_Property
  | OBE_Property AND_OP OBE_Property
  | OBE_Property OR_OP OBE_Property
  | OBE_Property **->** OBE_Property
  | OBE_Property **<->** OBE_Property

: Universal Operators :
  | **AX** OBE_Property
  | **AG** OBE_Property
  | **AF** OBE_Property
  | **A [** OBE_Property **U** OBE_Property **]**

```
: Existential Operators :
        | **EX** OBE_Property
        | **EG** OBE_Property
        | **EF** OBE_Property
        | **E [** OBE_Property **U** OBE_Property **]**
```

Appendix B

(informative)

Formal Syntax and Semantics of Accellera PSL

This appendix formally describes the syntax and semantics of the temporal layer.

B.1 Typed-text representation of symbols

Table 1 shows the mapping of various symbols used in this definition to the corresponding typed-text Sugar representation.

	Verilog	VHDL	EDL
$\mapsto$	`|->`	`|->`	`|->`
$\Mapsto$	`|=>`	`|=>`	`|=>`
$\rightarrow$	`->`	`->`	`->`
$\leftrightarrow$	`<->`	`<->`	`<->`
$\neg$	`!`	`not`	`!`
$\wedge$	`&&`	`and`	`&`
$\vee$	`||`	`or`	`|`
$..$	`:`	`to`	`..`
$(\,)$	`[ ]`	`( )`	`( )`

Table 1. Typed-text symbols in the Verilog, VHDL, and EDL flavors

Note:
For reasons of simplicity, the syntax given herein is more flexible than the one defined by the extended BNF (given in Appendix A). That is, some of the expressions which are legal here are not legal under the BNF Grammar. Users should use the stricter syntax, as defined by the BNF grammar in Appendix A.

B.2 Syntax

The logic Accellera PSL is defined with respect to a non-empty set of atomic propositions P and a given set of boolean expressions B over P. We assume two designated boolean expression *true* and *false* belong to B.

Definition 1 (Sequential Extended Regular Expressions (SEREs)).

– *Every boolean expression $b \in B$ is a SERE.*

– *If r, r_1, and r_2 are SEREs, and c is a boolean expression, then the following are SEREs:*
  - $\{r\}$
  - $r_1\ ;\ r_2$
  - $r_1 : r_2$
  - $r_1 \mid r_2$
  - $r_1\ \&\&\ r_2$
  - $[*0]$
  - $r[*]$
  - $r@c$

Definition 2 (Formulas of the Foundation Language (FL formulas)).

– *If b is a boolean expression then both b and $b!$ are FL formulas*[1].
– *If φ and ψ are FL formulas, r, r_1, r_2 are SEREs, and b a boolean expression, then the following are FL formulas:*
  - (φ)
  - $\neg\varphi$
  - $\varphi \wedge \psi$
  - $r!$
  - r
  - $X!\ \varphi$
  - $[\varphi\ U\ \psi]$
  - $\varphi\ \mathsf{abort}\ b$
  - $r \mapsto \varphi$
  - $\varphi@b$

Definition 3 (Formulas of the Optional Branching Extension (OBE)).

– *Every boolean expression is an OBE formula.*
– *If f, f_1, and f_2 are OBE formulas, then so are the following:*
  - (f)
  - $\neg f$
  - $f_1 \wedge f_2$
  - $EX f$
  - $E[f_1\ U\ f_2]$
  - $EG f$

Additional OBE operators are derived from these as follows:

– $f_1 \vee f_2 = \neg(\neg f_1 \wedge \neg f_2)$
– $f_1 \rightarrow f_2 = \neg f_1 \vee f_2$
– $f_1 \leftrightarrow f_2 = (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$
– $EF f = E[\mathrm{true}\ U\ f]$
– $AX f = \neg EX \neg f$
– $A[f_1\ U\ f_2] = \neg(E[\neg f_2\ U\ (\neg f_1 \wedge \neg f_2)] \vee EG\neg f_2)$
– $AG f = \neg E[\mathrm{true}\ U\ \neg f]$
– $AF f = A[\mathrm{true}\ U\ f]$

Definition 4 (Accellera PSL Formulas).

– *Every FL formula is an Accellera PSL formula.*
– *Every OBE formula is an Accellera PSL formula.*

In Section B.3, we show additional operators which provide syntactic sugaring to the ones above.

[1] We define formal semantics for both strong and weak booleans [2]. However, strong booleans are not accessible to the user.

B.3 Semantics

B.3.1 Semantics of FL formulas

The semantics of FL is defined with respect to finite and infinite words over $\Sigma = 2^P \cup \{\top, \bot\}$. We denote a letter from Σ by ℓ and an empty, finite, or infinite word from Σ by u, v, or w (possibly with subscripts). We denote the length of word v as $|v|$. An empty word $v = \epsilon$ has length 0, a finite word $v = (\ell_0\ell_1\ell_2 \cdots \ell_n)$ has length $n+1$, and an infinite word has length ∞. We use i, j, and k to denote non-negative integers. We denote the i^{th} letter of v by v^{i-1} (since counting of letters starts at zero). We denote by $v^{i..}$ the suffix of v starting at v^i. That is, for every $i < |v|$, $v^{i..} = v^i v^{i+1} \cdots v^n$ or $v^{i..} = v^i v^{i+1} \cdots$. We denote by $v^{i..j}$ the finite sequence of letters starting from v^i and ending in v^j. That is, for $j \geq i$, $v^{i..j} = v^i v^{i+1} \cdots v^j$ and for $j < i$, $v^{i..j} = \epsilon$. We use ℓ^ω to denote an infinite-length word, each letter of which is ℓ.

We use $\overline{v}$ to denote the word obtained by replacing every $\top$ with a $\bot$ and vice versa. We call $\overline{v}$ the *complement* of v.

The semantics of FL *formulas* over *words* is defined inductively, using as the base case the semantics of *boolean expressions* over *letters* in Σ. The semantics of boolean expression is assumed to be given as a relation $\Vdash\ \subseteq \Sigma \times B$ relating letters in Σ with boolean expressions in B. If $(\ell, b) \in\ \Vdash$ we say that the letter ℓ *satisfies* the boolean expression b and denote it $\ell \Vdash b$. We assume the two special letters $\top$ and $\bot$ behave as follows: for every boolean expression b, $\top \Vdash b$ and $\bot \not\Vdash b$. We assume that otherwise the boolean relation $\Vdash$ behaves in the usual manner. In particular, that for every letter $\ell \in 2^P$, atomic proposition $p \in P$ and boolean expressions $b, b_1, b_2 \in B$ (i) $\ell \Vdash p$ iff $p \in \ell$, (ii) $\ell \Vdash \neg b$ iff $\ell \not\Vdash b$, and (iii) $\ell \Vdash$ *true* and $\ell \not\Vdash$ *false*. Finally, we assume that for every letter $\ell \in \Sigma$, $\ell \Vdash b_1 \wedge b_2$ iff $\ell \Vdash b_1$ and $\ell \Vdash b_2$.

B.3.1.1 Unclocked Semantics

B.3.1.1.1 Semantics of unclocked SEREs

Unclocked SEREs are defined over finite words from the alphabet Σ. The notation $v \models\!\equiv r$, where r is a SERE and v a finite word means that v *models tightly* r. The semantics of unclocked SEREs are defined as follows, where b denotes a boolean expression, and r, r_1, and r_2 denote unclocked SEREs.

- $v \models\!\equiv \{r\} \iff v \models\!\equiv r$
- $v \models\!\equiv b \iff |v| = 1$ and $v^0 \Vdash b$
- $v \models\!\equiv r_1 \ ;\ r_2 \iff \exists v_1, v_2$ s.t. $v = v_1 v_2$, $v_1 \models\!\equiv r_1$, and $v_2 \models\!\equiv r_2$
- $v \models\!\equiv r_1 : r_2 \iff \exists v_1, v_2$, and ℓ s.t. $v = v_1 \ell v_2$, $v_1\ell \models\!\equiv r_1$, and $\ell v_2 \models\!\equiv r_2$
- $v \models\!\equiv r_1 \mid r_2 \iff v \models\!\equiv r_1$ or $v \models\!\equiv r_2$
- $v \models\!\equiv r_1$ && $r_2 \iff v \models\!\equiv r_1$ and $v \models\!\equiv r_2$
- $v \models\!\equiv [*0] \iff v = \epsilon$
- $v \models\!\equiv r[*] \iff$ either $v \models\!\equiv [*0]$ or $\exists v_1, v_2$ s.t. $v_1 \neq \epsilon$, $v = v_1 v_2$, $v_1 \models\!\equiv r$ and $v_2 \models\!\equiv r[*]$

B.3.1.1.2 Semantics of unclocked FL

We refer to a formula of FL with no @ operator as an *unclocked formula*. Let v be a finite or infinite word, b be a boolean expression, r, r_1, r_2 unclocked SEREs, and φ, ψ unclocked FL formulas. We use $\models$ to define the semantics of unclocked FL formulas: If $v \models \varphi$ we say that v *models* (or *satisfies*) φ.

1. $v \models (\varphi) \iff v \models \varphi$
2. $v \models \neg\varphi \iff \overline{v} \not\models \varphi$
3. $v \models \varphi \wedge \psi \iff v \models \varphi$ and $v \models \psi$
4. $v \models b! \iff |v| > 0$ and $v^0 \vDash b$
5. $v \models b \iff |v| = 0$ or $v^0 \vDash b$
6. $v \models r! \iff \exists j < |v|$ s.t. $v^{0..j} \mid\!\equiv r$
7. $v \models r \iff \forall j < |v|$, $v^{0..j}\top^\omega \models r!$
8. $v \models X!\ \varphi \iff |v| > 1$ and $v^{1..} \models \varphi$
9. $v \models [\varphi U \psi] \iff \exists k < |v|$ s.t. $v^{k..} \models \psi$, and $\forall j < k$, $v^{j..} \models \varphi$
10. $v \models \varphi$ abort $b \iff$ either $v \models \varphi$ or $\exists j < |v|$ s.t. $v^j \vDash b$ and $v^{0..j-1}\top^\omega \models \varphi$
11. $v \models r \mapsto \varphi \iff \forall j < |v|$ s.t. $\overline{v}^{0..j} \mid\!\equiv r$, $v^{j..} \models \varphi$

Notes:

1. The semantics given here for the LTL operator and the abort operator is equivalent to the truncated semantics given in [1] which is interpreted over 2^P rather than over $2^P \cup \{\top, \bot\}$. Using $\models_\bullet$ for the semantics in [1] the following proposition states the equivalence: Let w be a finite word over 2^P, and let φ be a formula of LTL^{trunc}. Then the three following equivalences hold:
$$w \overset{-}{\models}_\bullet \varphi \iff w\top^\omega \models \varphi$$
$$w \models_\bullet \varphi \iff w \models \varphi$$
$$w \overset{+}{\models}_\bullet \varphi \iff w\bot^\omega \models \varphi$$
2. Using $\models_\bullet$ as in the note 1 above, we use *holds strongly* for $\overset{+}{\models}_\bullet$, *holds* for $\models_\bullet$, and *holds weakly* for $\overset{-}{\models}_\bullet$. The remaining terminology of Section 4.4.6 is formally defined as follows:
   - φ is *pending* on word w iff $w \overset{-}{\models}_\bullet \varphi$ and $w \not\models_\bullet \varphi$
   - φ *fails* on word w iff $w \not\models_\bullet \varphi$
3. There is a subtle difference between boolean negation and formula negation. For instance, consider the formula $\neg b$. If $\neg$ is boolean negation, then $\neg b$ holds on an empty path. If $\neg$ is formula negation, then $\neg b$ does not hold on an empty path. Rather than introduce distinct operators for boolean and formula negation, we instead adopt the convention that negation applied to a boolean expression is boolean negation. This does not restrict expressivity, as formula negation of b can be expressed as $(\neg b)!$.

B.3.1.2 Clocked Semantics

We say that finite word v *is a clock tick of* c iff $|v| > 0$ and $v^{|v|-1} \Vdash c$ and for every natural number $i < |v| - 1$, $v^i \Vdash \neg c$.

B.3.1.2.1 Semantics of clocked SEREs

Clocked SEREs are defined over finite words from the alphabet Σ and a boolean expression that serves as the clock context. The notation $v \mathrel{|\overset{c}{\equiv}} r$, where r is a SERE and c is a boolean expression, means that v *models tightly* r *in context of clock* c. The semantics of clocked SEREs are defined as follows, where b, c, and c_1 denote boolean expressions, r, r_1, and r_2 denote clocked SEREs.

- $v \mathrel{|\overset{c}{\equiv}} \{r\} \iff v \mathrel{|\overset{c}{\equiv}} r$
- $v \mathrel{|\overset{c}{\equiv}} b \iff v$ is a clock tick of c and $v^{|v|-1} \Vdash b$
- $v \mathrel{|\overset{c}{\equiv}} r_1 \ ; \ r_2 \iff \exists v_1, v_2$ s.t. $v = v_1 v_2$, $v_1 \mathrel{|\overset{c}{\equiv}} r_1$, and $v_2 \mathrel{|\overset{c}{\equiv}} r_2$
- $v \mathrel{|\overset{c}{\equiv}} r_1 : r_2 \iff \exists v_1, v_2$, and ℓ s.t. $v = v_1 \ell v_2$, $v_1 \ell \mathrel{|\overset{c}{\equiv}} r_1$, and $\ell v_2 \mathrel{|\overset{c}{\equiv}} r_2$
- $v \mathrel{|\overset{c}{\equiv}} r_1 \mid r_2 \iff v \mathrel{|\overset{c}{\equiv}} r_1$ or $v \mathrel{|\overset{c}{\equiv}} r_2$
- $v \mathrel{|\overset{c}{\equiv}} r_1 \ \&\& \ r_2 \iff v \mathrel{|\overset{c}{\equiv}} r_1$ and $v \mathrel{|\overset{c}{\equiv}} r_2$
- $v \mathrel{|\overset{c}{\equiv}} [*0] \iff v = \epsilon$
- $v \mathrel{|\overset{c}{\equiv}} r[*] \iff$ either $v \mathrel{|\overset{c}{\equiv}} [*0]$ or $\exists v_1, v_2$ s.t. $v_1 \neq \epsilon$, $v = v_1 v_2$, $v_1 \mathrel{|\overset{c}{\equiv}} r$ and $v_2 \mathrel{|\overset{c}{\equiv}} r[*]$
- $v \mathrel{|\overset{c}{\equiv}} r@c_1 \iff v \mathrel{|\overset{c_1}{\equiv}} r$

B.3.1.2.2 Semantics of clocked FL

The semantics of (clocked) FL formulas is defined with respect to finite/infinite words over Σ and a boolean expression c which serves as the clock context. Let v be a finite or infinite word, b, c, c_1 boolean expressions, r, r_1, r_2 SEREs, and φ, ψ FL formulas. We use $\overset{c}{\models}$ to define the semantics of FL formulas. If $v \overset{c}{\models} \varphi$ we say that v *models* (or *satisfies*) φ *in the context of clock* c.

1. $v \overset{c}{\models} (\varphi) \iff v \overset{c}{\models} \varphi$
2. $v \overset{c}{\models} \neg\varphi \iff \overline{v} \overset{c}{\not\models} \varphi$
3. $v \overset{c}{\models} \varphi \wedge \psi \iff v \overset{c}{\models} \varphi$ and $v \overset{c}{\models} \psi$
4. $v \overset{c}{\models} b! \iff \exists j < |v|$ s.t. $v^{0..j}$ is a clock tick of c and $v^j \Vdash b$
5. $v \overset{c}{\models} b \iff \forall j < |v|$ s.t. $\overline{v}^{0..j}$ is a clock tick of c, $v^j \Vdash b$
6. $v \overset{c}{\models} r! \iff \exists j < |v|$ s.t. $v^{0..j} \mathrel{|\overset{c}{\equiv}} r$
7. $v \overset{c}{\models} r \iff \forall j < |v|$, $v^{0..j}\top^\omega \overset{c}{\models} r!$
8. $v \overset{c}{\models} X! \ f \iff \exists j < k < |v|$ s.t. $v^{0..j}$ and $v^{j+1..k}$ are clock ticks of c and $v^{k..} \overset{c}{\models} f$
9. $v \overset{c}{\models} [\varphi U \psi] \iff \exists k < |v|$ s.t. $v^k \Vdash c$, $v^{k..} \overset{c}{\models} \psi$, and $\forall j < k$ s.t. $v^j \Vdash c$, $v^{j..} \overset{c}{\models} \varphi$
10. $v \overset{c}{\models} \varphi$ abort $b \iff$ either $v \overset{c}{\models} \varphi$ or $\exists j < |v|$ s.t. $v^j \Vdash b$ and $v^{0..j-1}\top^\omega \overset{c}{\models} \varphi$

11. $v \models^{c} r \mapsto \varphi \iff \forall j < |v|$ s.t. $\overline{v}^{0..j} \models^{c} r$, $v^{j..} \models^{c} \varphi$
12. $v \models^{c} \varphi @ c_1 \iff v \models^{c_1} \varphi$

Note:
The clocked semantics for the LTL subset follows the clocks paper [2], with the exception that strength is applied at the boolean level rather than at the propositional level.

B.2.2 Semantics of OBE formulas

The semantics of OBE formulas are defined over states in the *model*, rather than finite or infinite words. A model is a quintuple (S, S_0, R, P, L), where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is the transition relation, P is a non-empty set of atomic propositions, and L is the valuation, a function $L : S \longrightarrow 2^P$, mapping each state with a set of atomic propositions valid in that state.

A *path* π is a finite (or infinite) sequence of states $\pi = (\pi_0, \pi_1, \pi_2, \cdots, \pi_n)$ (or $\pi = (\pi_0, \pi_1, \pi_2, \cdots)$). A *computation path* π of a model M is a finite (or infinite) path π such that for every $i < n$, $R(\pi_i, \pi_{i+1})$ and for no s, $R(\pi_n, s)$ (or such that for every i, $R(\pi_i, \pi_{i+1})$). Given a finite (or infinite) path π, we define $\hat{L}$, an extension of the valuation function L from states to paths as follows: $\hat{L}(\pi) = L(\pi_0)L(\pi_1)\ldots L(\pi_n)$ (or $\hat{L}(\pi) = L(\pi_0)L(\pi_1)\ldots$). Thus we have a mapping from states in M to letters of 2^P, and from finite (or infinite) sequences of states in M to finite (or infinite) words over 2^P.

The semantics of OBE formulas are defined inductively, using as the base case the semantics of *boolean expressions* over *letters* in 2^P. The semantics of boolean expression is assumed to be given as a relation $\Vdash \subseteq 2^P \times B$ relating letters in 2^P with boolean expressions in B. If $(\ell, b) \in \Vdash$ we say that the letter ℓ *satisfies* the boolean expression b and denote it $\ell \Vdash b$. We assume that the boolean relation $\Vdash$ behaves in the usual manner. In particular, that for every letter $\ell \in 2^P$, atomic proposition $p \in P$ and boolean expressions $b, b_1, b_2 \in B$ (i) $\ell \Vdash p$ iff $p \in \ell$, (ii) $\ell \Vdash \neg b$ iff $\ell \nVdash b$, (iii) $\ell \Vdash b_1 \wedge b_2$ iff $\ell \Vdash b_1$ and $\ell \Vdash b_2$, and (iv) $\ell \Vdash$ *true* and $\ell \nVdash$ *false*.

The notation $M, s \models f$ means that formula f holds in state s of model M. The notation $M \models f$ is equivalent to $\forall s \in S_0 : M, s \models f$. In other words, f is valid for every initial state of M. The semantics of an OBE formula are defined as follows[2], where b denotes a boolean expression and f, f_1, and f_2 denote OBE formulas.

- $M, s \models b \iff L(s) \Vdash b$
- $M, s \models (f) \iff M, s \models f$
- $M, s \models \neg f \iff M, s \not\models f$
- $M, s \models f_1 \wedge f_2 \iff M, s \models f_1$ and $M, s \models f_2$
- $M, s \models EX\ f \iff$ there exists a computation path π of M such that $|\pi| > 1$, $\pi_0 = s$, and $M, \pi_1 \models f$

[2] The semantics are those of standard CTL.

- $M, s \models E[f_1\ U\ f_2] \iff$ there exists a computation path π of M such that $\pi_0 = s$ and there exists $k < |\pi|$ such that $M, \pi_k \models f_2$ and for every j such that $j < k$: $M, \pi_j \models f_1$
- $M, s \models EG\ f \iff$ there exists a computation path π of M such that $\pi_0 = s$ and for every j such that $0 \leq j < |\pi|$: $M, \pi_j \models f$

B.4 Syntactic Sugaring

The remainder of the temporal layer is syntactic sugar. In other words, it does not add expressive power, and every piece of syntactic sugar can be defined in terms of the basic FL operators presented above. The syntactic sugar is defined below.

Note: the definitions given here do not necessarily represent the most efficient implementation. In some cases, there is an equivalent syntactic sugaring, or a direct implementation, that is more efficient.

B.3.1 Additional SERE operators

If i, j, k, and l are integer constants such that $i \geq 0$, $j \geq i$, $k \geq 1$ and $l \geq k$, then additional SERE operators can be viewed as abbreviations of the basic SERE operators defined above, as follows, where b denotes a boolean expression, and r denotes a SERE.

- $r[+] \overset{\text{def}}{=} r; r[*]$
- $r[*0] \overset{\text{def}}{=} [*0]$
- $r[*k] \overset{\text{def}}{=} \overbrace{r; r; \ldots; r}^{k\ times}$
- $r[*i..j] \overset{\text{def}}{=} r[*i] \mid \ldots \mid r[*j]$
- $r[*i..] \overset{\text{def}}{=} r[*i]; r[*]$
- $r[*..i] \overset{\text{def}}{=} r[*0] \mid \ldots \mid r[*i]$
- $r[*..] \overset{\text{def}}{=} r[*0..]$
- $[+] \overset{\text{def}}{=} true[+]$
- $[*] \overset{\text{def}}{=} true[*]$
- $[*i] \overset{\text{def}}{=} true[*i]$
- $[*i..j] \overset{\text{def}}{=} true[*i..j]$
- $[*i..] \overset{\text{def}}{=} true[*i..]$
- $[*..i] \overset{\text{def}}{=} true[*..i]$
- $[*..] \overset{\text{def}}{=} true[*..]$
- $b[=i] \overset{\text{def}}{=} \{\neg b[*]; b\}[*i]; \neg b[*]$
- $b[=i..j] \overset{\text{def}}{=} b[=i] \mid \ldots \mid b[=j]$
- $b[=i..] \overset{\text{def}}{=} b[=i]; [*]$

- $b[= ..i] \overset{\text{def}}{=} b[= 0] \mid ... \mid b[= i]$
- $b[= ..] \overset{\text{def}}{=} b[= 0..]$
- $b[\rightarrow] \overset{\text{def}}{=} \neg b[*]; b$
- $b[\rightarrow k] \overset{\text{def}}{=} \{\neg b[*]; b\}[*k]$
- $b[\rightarrow k..l] \overset{\text{def}}{=} b[\rightarrow k] \mid ... \mid b[\rightarrow l]$
- $b[\rightarrow k..] \overset{\text{def}}{=} b[\rightarrow k] \mid \{b[\rightarrow k]; [*]; b\}$
- $b[\rightarrow ..k] \overset{\text{def}}{=} b[\rightarrow 1] \mid ... \mid b[\rightarrow k]$
- $b[\rightarrow ..] \overset{\text{def}}{=} b[\rightarrow 1..]$
- $r_1 \,\&\, r_2 \overset{\text{def}}{=} \{\{r_1\} \,\&\&\, \{r_2; true[*]\}\} \mid \{\{r_1; true[*]\} \,\&\&\, \{r_2\}\}$
- $r_1 \ within \ r_2 \overset{\text{def}}{=} \{[*];\ r_1;\ [*]\} \,\&\&\, \{r_2\}$

B.3.2 Additional FL operators

If i, j, k and l are integers such that $i \geq 0$, $j \geq i$, $k > 0$ and $l \geq k$ then additional operators can be viewed as abbreviations of the basic operators defined above, as follows, where b denotes a boolean expression, r, r_1, and r_2 denote SEREs, and φ, φ_1, and φ_2 denote FL formulas.

- $\varphi_1 \vee \varphi_2 \overset{\text{def}}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2)$
- $\varphi_1 \rightarrow \varphi_2 \overset{\text{def}}{=} \neg\varphi_1 \vee \varphi_2$
- $\varphi_1 \leftrightarrow \varphi_2 \overset{\text{def}}{=} (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$

- $F\varphi \overset{\text{def}}{=} [true\ U\ \varphi]$
- $G\varphi \overset{\text{def}}{=} \neg F \neg\varphi$
- $X\varphi \overset{\text{def}}{=} \neg X!\ \neg\varphi$
- $[\varphi_1\ W\ \varphi_2] \overset{\text{def}}{=} [\varphi_1\ U\ \varphi_2] \vee G\varphi_1$

- $always\ \varphi \overset{\text{def}}{=} G\ \varphi$
- $never\ \varphi \overset{\text{def}}{=} G\ \neg\varphi$
- $next!\ \varphi \overset{\text{def}}{=} X!\ \varphi$
- $next\ \varphi \overset{\text{def}}{=} X\ \varphi$
- $eventually!\ \varphi \overset{\text{def}}{=} F\varphi$

- $\varphi_1\ until!\ \varphi_2 \overset{\text{def}}{=} [\varphi_1\ U\ \varphi_2]$
- $\varphi_1\ until\ \varphi_2 \overset{\text{def}}{=} [\varphi_1\ W\ \varphi_2]$
- $\varphi_1\ until!\_\ \varphi_2 \overset{\text{def}}{=} [\varphi_1\ U\ \varphi_1 \wedge \varphi_2]$

- $\varphi_1\ until\_\ \varphi_2 \overset{\text{def}}{=} [\varphi_1\ W\ \varphi_1 \wedge \varphi_2]$

- $\varphi_1\ before!\ \varphi_2 \overset{\text{def}}{=} [\neg\varphi_2\ U\ \varphi_1 \wedge \neg\varphi_2]$
- $\varphi_1\ before\ \varphi_2 \overset{\text{def}}{=} [\neg\varphi_2\ W\ \varphi_1 \wedge \neg\varphi_2]$
- $\varphi_1\ before!\_\ \varphi_2 \overset{\text{def}}{=} [\neg\varphi_2\ U\ \varphi_1]$
- $\varphi_1\ before\_\ \varphi_2 \overset{\text{def}}{=} [\neg\varphi_2\ W\ \varphi_1]$

- $X!\ [i]\varphi \overset{\text{def}}{=} \overbrace{X!\ X!\ ...X!}^{i\ times}\ \varphi$
- $X[i]\varphi \overset{\text{def}}{=} \overbrace{XX...X}^{i\ times}\ \varphi$
- $next![i]\ \varphi \overset{\text{def}}{=} X!\ [i]\ \varphi$
- $next[i]\ \varphi \overset{\text{def}}{=} X[i]\ \varphi$
- $next\_a![i..j]\varphi \overset{\text{def}}{=} (X![i]\varphi) \wedge \ldots \wedge (X![j]\varphi)$
- $next\_a[i..j]\varphi \overset{\text{def}}{=} (X[i]\varphi) \wedge \ldots \wedge (X[j]\varphi)$
- $next\_e![i..j]\varphi \overset{\text{def}}{=} (X![i]\varphi) \vee \ldots \vee (X![j]\varphi)$
- $next\_e[i..j]\varphi \overset{\text{def}}{=} (X[i]\varphi) \vee \ldots \vee (X[j]\varphi)$

- $next\_event!(b)(\varphi) \overset{\text{def}}{=} [\neg b\ U\ b \wedge \varphi]$
- $next\_event(b)(\varphi) \overset{\text{def}}{=} [\neg b\ W\ b \wedge \varphi]$
- $next\_event!(b)[k](\varphi) \overset{\text{def}}{=} next\_event!(b)\ \overbrace{(X!\ next\_event!(b)...(X!\ next\_event!(b)}^{k-1\ times}(\varphi))...)$
- $next\_event(b)[k](\varphi) \overset{\text{def}}{=} next\_event(b)\ \overbrace{(X next\_event(b)...(X next\_event(b)}^{k-1\ times}(\varphi))...)$
- $next\_event\_a!(b)[k..l](\varphi) \overset{\text{def}}{=} next\_event!(b)[k](\varphi) \wedge ... \wedge next\_event!(b)[l](\varphi)$
- $next\_event\_a(b)[k..l](\varphi) \overset{\text{def}}{=} next\_event(b)[k](\varphi) \wedge ... \wedge next\_event(b)[l](\varphi)$
- $next\_event\_e!(b)[k..l](\varphi) \overset{\text{def}}{=} next\_event!(b)[k](\varphi) \vee ... \vee next\_event!(b)[l](\varphi)$
- $next\_event\_e(b)[k..l](\varphi) \overset{\text{def}}{=} next\_event(b)[k](\varphi) \vee .... \vee next\_event(b)[l](\varphi)$

- $r(\varphi) \overset{\text{def}}{=} r \mapsto \varphi$
- $r \Rightarrow \varphi \overset{\text{def}}{=} \{r; true\} \mapsto \varphi$

B.3.4 Forall

If f is an Accellera PSL formula, $v_0, v_1, \cdots, v_n$ are constants, and j, k, l and m are integers, then the following are Accellera PSL formulas:

- $forall\ i\ in\ \{v_0, v_1, \cdots, v_n\} : f$

- $forall\ i\ in\ j..k : f$
- $forall\ i\ in\ boolean : f$
- $forall\ i\langle l..m\rangle\ in\ \{v_0, v_1, \cdots, v_n\} : f$
- $forall\ i\langle l..m\rangle\ in\ j..k : f$
- $forall\ i\langle l..m\rangle\ in\ boolean : f$

Forall does not add expressive power. Rather, it can be viewed as additional syntactic sugar, as follows:

- $forall\ i\ in\ \{v_0, v_1, \cdots, v_n\} : f \stackrel{\text{def}}{=} \bigwedge_{u \in \{v_0, v_1, \cdots, v_n\}} f[i \leftarrow u]$
- $forall\ i\ in\ j..k : f \stackrel{\text{def}}{=} \bigwedge_{u=j}^{k} f[i \leftarrow u]$
- $forall\ i\ in\ boolean : f \stackrel{\text{def}}{=} \bigwedge_{u=0}^{1} f[i \leftarrow u]$
- $forall\ i\langle l..m\rangle\ in\ \{v_0, v_1, \cdots, v_n\} : f \stackrel{\text{def}}{=} \bigwedge_{u_l \in \{v_0, v_1, \cdots, v_n\}} \cdots \bigwedge_{u_m \in \{v_0, v_1, \cdots, v_n\}} f[i\langle l..m\rangle \leftarrow \langle u_l..u_m\rangle]$
- $forall\ i\langle l..m\rangle\ in\ j..k : f \stackrel{\text{def}}{=} \bigwedge_{u_l=j}^{k} \cdots \bigwedge_{u_m=j}^{k} f[i\langle l..m\rangle \leftarrow \langle u_l..u_m\rangle]$
- $forall\ i\langle l..m\rangle\ in\ boolean : f \stackrel{\text{def}}{=} \bigwedge_{u_l=0}^{1} \cdots \bigwedge_{u_m=0}^{1} f[i\langle l...m\rangle \leftarrow \langle u_l..u_m\rangle]$

where $f[i \leftarrow u]$ is the formula obtained from f by replacing every occurrence of i by u and $f[i\langle l..m\rangle \leftarrow \langle u_l..u_m\rangle]$ is the formula obtained from f by replacing every occurrence of i_j with u_j.

B.5 Rewriting rules for clocks

In Section B.2.2 we gave the semantics of clocked FL formulas directly. There is an equivalent definition in terms of unclocked FL formulas, as follows: Starting from the outermost clock, use the following rules to translate clocked SEREs into unclocked SEREs, and clocked FL formulas into unclocked FL formulas.

The rewrite rules for SEREs are:

1. $\mathcal{R}^c(\{r\}) = \mathcal{R}^c(r)$
2. $\mathcal{R}^c(b) = \neg c[*]; c \wedge b$
3. $\mathcal{R}^c(r_1 ; r_2) = \mathcal{R}^c(r_1) ; \mathcal{R}^c(r_2)$

4. $\mathcal{R}^c(r_1 : r_2) = \{\mathcal{R}^c(r_1)\} : \{\mathcal{R}^c(r_2)\}$
5. $\mathcal{R}^c(r_1 \mid r_2) = \{\mathcal{R}^c(r_1)\} \mid \{\mathcal{R}^c(r_2)\}$
6. $\mathcal{R}^c(r_1 \mathbin{\&\&} r_2) = \{\mathcal{R}^c(r_1)\} \mathbin{\&\&} \{\mathcal{R}^c(r_2)\}$
7. $\mathcal{R}^c([*0]) = [*0]$
8. $\mathcal{R}^c(r[*]) = \{\mathcal{R}^c(r)\}[*]$
9. $\mathcal{R}^c(r@c_1) = \mathcal{R}^{c_1}(r)$

The rewrite rules for FL formulas are:

1. $\mathcal{F}^c((\varphi)) = (\mathcal{F}^c(\varphi))$
2. $\mathcal{F}^c(b!) = [\neg c \ U \ (c \wedge b)]$
3. $\mathcal{F}^c(b) = [\neg c \ W \ (c \wedge b)]$
4. $\mathcal{F}^c(\neg\varphi) = \neg\mathcal{F}^c(\varphi)$
5. $\mathcal{F}^c(\varphi \wedge \psi) = (\mathcal{F}^c(\varphi) \wedge \mathcal{F}^c(\psi))$
6. $\mathcal{F}^c(X!\varphi) = [\neg c \ U \ (c \wedge X! \ [\neg c \ U \ (c \wedge \mathcal{F}^c(\varphi))])]$
7. $\mathcal{F}^c(\varphi \ U \ \psi) = [(c \rightarrow \mathcal{F}^c(\varphi)) \ U \ (c \wedge \mathcal{F}^c(\psi))]$
8. $\mathcal{F}^c(\varphi \ \mathsf{abort} \ b) = \mathcal{F}^c(\varphi) \ \mathsf{abort} \ b$
9. $\mathcal{F}^c(\varphi@c_1) = \mathcal{F}^{c_1}(\varphi)$
10. $\mathcal{F}^c(r \mapsto \varphi) = \mathcal{R}^c(r) \mapsto \mathcal{F}^c(\varphi)$
11. $\mathcal{F}^c(r!) = \mathcal{R}^c(r)!$
12. $\mathcal{F}^c(r) = \mathcal{R}^c(r)$

NOTE: The v1.1 formal semantics presented here correct the three anomalies described in Section B.6 of the LRM v1.0. An additional anomaly, since discovered, is not yet corrected. It is as follows: the *logical contradiction false* is considered to be weakly satisfiable, but the *structural contradiction* $\{\{a\}\&\&\{a;a\}\}$ is not. For instance, the property $F false$ holds weakly on a finite path, but the property $F\{\{a\}\&\&\{a;a\}\}$ does not. This issue will be addressed in the next version of the formal semantics. From the user's point of view, this will have minimal effect, since it is a corner case resulting from the use of a non-satisfiable SERE. From a tool builder's point of view, this will have minimal effect since the change involves removing one step in the algorithm that builds the automaton for a given SERE (the step that removes states from which there is no accepting run).

References

1. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In *The 15th International Conference on Computer Aided Verification (CAV'03)*, LNCS 2725, pages 27–40, Boulder, CO, USA, July 2003. Springer-Verlag.
2. C. Eisner, D. Fisman, J. Havlicek, A. McIsaac, and D. Van Campenhout. The definition of a temporal clock operator. In *Proc. 30th Int. Colloq. Aut. Lang. Prog. (ICALP'03)*, LNCS 2719, pages 857–870. Springer-Verlag, June 2003.