Property Specification Language

Reference Manual

Version 1.0

January 31, 2003

Additional copies of this manual may be purchased by contacting Accellera at the address shown below.

Notices

The information contained in this manual represents the definition of the Property Specification Language as reviewed and released by Accellera in January 2003.

Accellera reserves the right to make changes to the Property Specification Language and this manual in subsequent revisions and makes no warranties whatsoever with respect to the completeness, accuracy, or applicability of the information in this manual, when used for production design and/or development.

Accellera does not endorse any particular simulator or other CAE tool that is based on the Property Specification Language.

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

or to the address below.

The current Working Group's website address is

www.eda.org/vfv

Information about Accellera and membership enrollment can be obtained by inquiring at the address below.

Published as: Property Specification Language Reference Manual
Version 1.0, January 31, 2003.

Published by: Accellera
1370 Trancas Street, #163
Napa, CA 94558
Phone: (707) 251-9977
Fax: (707) 251-9877

Printed in the United States of America.

The following individuals contributed to the creation, editing, and review of *Property Specification Language* 1.0

Ken Albin	Motorola, Inc.	
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	Department of Computer Science, Carnegie Mellon	
Joe Daniels		Technical Editor
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	
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	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.	
Richard Ho	0-In Design Automation, Inc.	
Yaron Kashai	Verisity Design, Inc.	
Joseph Lu	Sun Microsystems	
Adriana Maggiore	TransEDA Technology Ltd	
Erich Marschner	Cadence Design Systems	Work Group Co-Chair
Anthony McIsaac	STMicroelectronics, Ltd.	
Hillel Miller	Motorola, Inc.	
Carl Pixley	Synopsys, Inc.	
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.	
Moshe Y. Vardi	Rice University	
Bow-Yaw Wang	Verplex Systems, Inc.	
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

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 which 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 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, the Boolean expression

```
ena || enb
```

describes a cycle in which one of the signals `ena` and `enb` are asserted. The 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 which has the functionality of the Finite State Machine (FSM) shown in Figure 1.

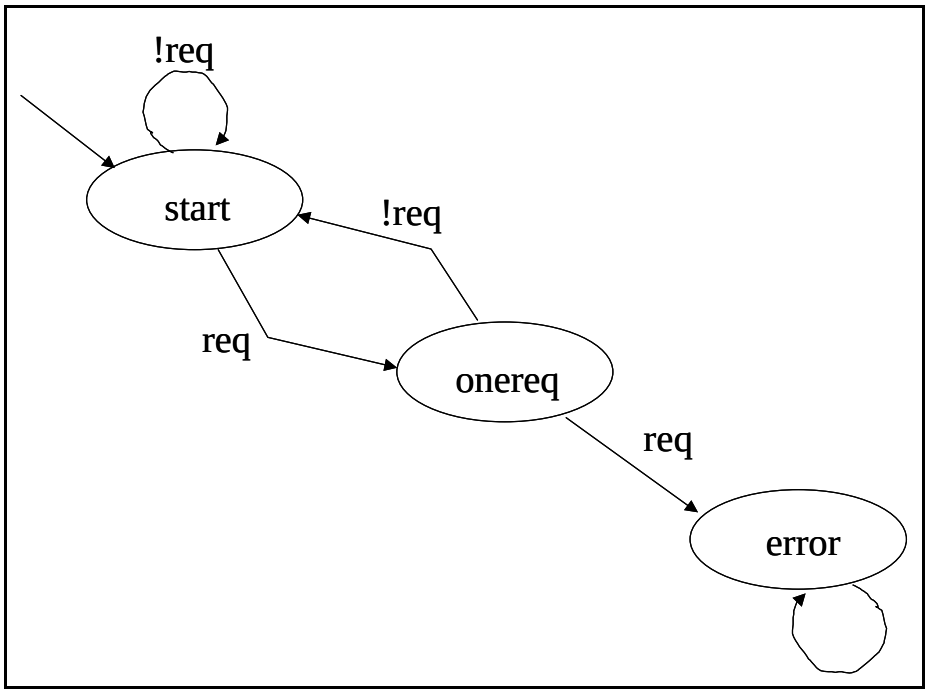

Figure 1—A simple (deterministic) FSM which checks the above property

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 which automatically generates simulatable checkers.

While all PSL properties can be in principle 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 4.4.5 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.

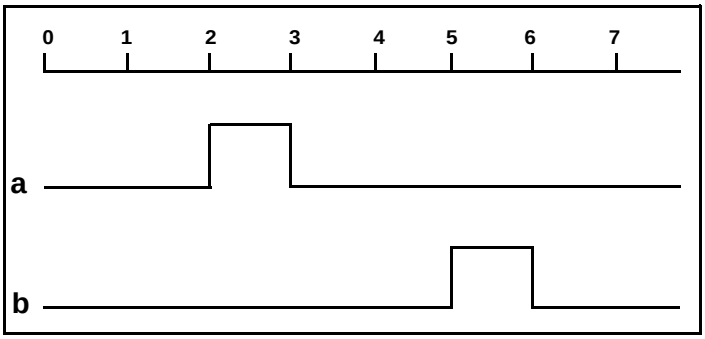

Figure 2—A trace which satisfies "always (a -> next[3] b)"

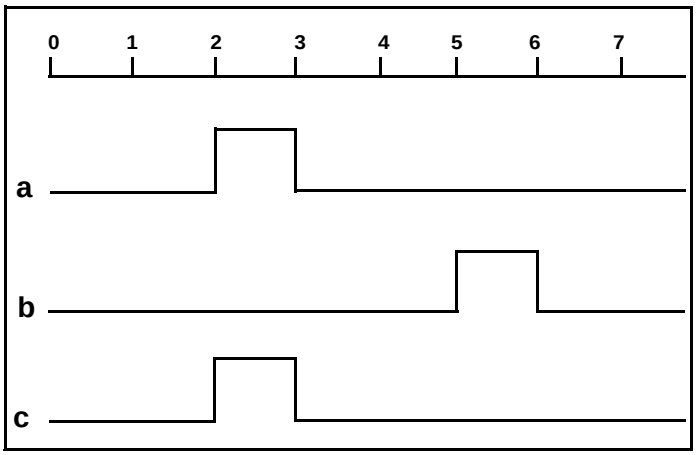

Figure 3—A trace which satisfies "always ((a & next[3] b) -> c)"

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.[1]
— Appendix C (Bibliography) provides additional documents, to which reference is made only for information or background purposes.

[1] The Accellera Property Specification Language is based upon the Sugar 2.0 property specification language. Appendix B presents the formal syntax and semantics of Sugar 2.0, which in turn defines the formal syntax and semantics of the temporal layer of PSL. Specifically, the formulas of the Sugar Foundation Language define the syntax and semantics of properties of the PSL Foundation Language, and the formulas of the (Sugar) Optional Branching Extension define the syntax and semantics of properties of the PSL Optional Branching Extension.

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.

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.

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) which limits the set of behavior 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 behavior 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 behavior 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 Verilog flavor, the single bit value `1'b0` is interpreted as the logical value *False*. In the VHDL flavor, the values `STD.Standard.Boolean'(False)`, `STD.Standard.Bit'('0')`, and `IEEE.std_logic_1164.std_logic'('0')` are all interpreted as the logical value *False*. In the EDL 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 behavior 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 behavior, 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 both logical values and metalogical 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 **metalogical value:** A value of a (multi-valued) logic type that is not interpreted as a logical value.

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

3.1.30 **monitor:** *See:* **checker**.

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

3.1.32 **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.

3.1.33 **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.

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

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

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

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

3.1.38 **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 behavior.

3.1.39 **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 the left expression specifies the low bound of the series, and the right expression specifies the high bound of the series.

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

3.1.41 **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.

3.1.42 **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.

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

3.1.44 **sequential expression:** A finite series of terms that represent a set of behavior.

3.1.45 **SERE:** A sequential expression.

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

3.1.47 **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 which 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.

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

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

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

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

3.1.52 **True:** An interpretation of certain values of certain data types in an HDL.
In the Verilog flavor, 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 EDL flavor, the Boolean value `'true'` and bit value `1B` are both interpreted as the logical value *True*.

3.1.53 **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.

3.1.54 **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
EDL	Environment Description Language
FL	Foundation Language
FSM	finite state machine
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	Sugar 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 three 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 which 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 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 three *flavors*: one for each of the hardware description languages Verilog, VHDL, and EDL. 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 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`.

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

4.1.2.2 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`.

4.1.2.3 EDL flavor

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

4.2 Lexical structure

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

4.2.1 Keywords

Keywords in PSL are case-sensitive, regardless of the underlying HDL rules for identifiers. Keywords are reserved words 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.

The keywords used in PSL are shown in Table 1.

Table 1—Keywords

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

[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).

4.2.2 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 in most cases the operators are associated with their operands in left-to-right order of appearance in the text, except for implication operators, which are associated with their operands in right-to-left order.

Table 2—Operator precedence

HDL operators	
Clocking operator	`@`
SERE construction operatora	`;  [* ]  [= ]  [-> ]`
Sequence implication operators	`:  |  &  &&`
FL implication operators	`|->  |-> !  |=>  |=> !`

FL occurence operators	`always  never  eventually!  next*`
	`within*  whilenot*  G  F`
	`X  X!  [ U ]  [ W ]`

Termination operators	`abort  until*  before*`

4.2.2.1 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, as if the PSL properties that are operands produced Boolean values of a type appropriate to the logical operators native to the HDL.

4.2.2.2 Foundation Language (FL) operators

4.2.2.2.1 Clocking operator

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

`@` clock event

The clocking operator is left-associative.

4.2.2.2.2 SERE construction operators

For any flavor of PSL, the Foundation Language (FL) operators with the next highest precedence are those used to construct Sugar Extended Regular Expressions (SEREs). These operators are:

`;`	temporal concatenation
`[* ]`	consecutive repetition
`[= ]`	non-consecutive repetition
`[-> ]`	goto repetition

SERE construction operators are left-associative.

4.2.2.2.3 Sequence composition operators

For any flavor of PSL, the FL operators with the next highest precedence are those used to compose sequences into longer or more complex descriptions of behavior. These operators are:

`:`	sequence fusion
`|`	sequence disjunction
`&`	non-length-matching sequence conjunction
`&&`	length-matching sequence conjunction

Sequence composition operators are left-associative.

4.2.2.2.4 FL implication operators

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

`|->`	weak suffix implication
`|-> !`	strong suffix implication
`|=>`	weak next suffix implication
`|=> !`	strong next suffix implication
`->`	logical IF implication
`<->`	logical IFF implication

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

NOTE—The syntax does not allow cascading of suffix implication operators.

4.2.2.2.5 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:

`always`	must hold, globally
`never`	must NOT hold, globally
`eventually!`	must hold at some time in the indefinite future
`next*`[3]	must hold at some specified future time or range of future times
`within*`[4]	must hold following completion of a sequence until a termination condition
`whilenot*`	must hold from the current cycle until a termination condition

[3]The notation `next*` represents the operators `next`, `next!`, `next_a`, `next_a!`, `next_e`, `next_e!`, `next_event`, `next_event!`, `next_event_a!`, and `next_event_e!`.

[4]The notation `within*` represents the operators `within`, `within!`, `within!_`, and `within_`. Similarly, `whilenot*`, `until*`, and `before*` each represent the corresponding family of operators.

FL occurrence operators are left-associative.

4.2.2.2.6 Termination operators

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

`abort`	must hold, but future obligations may be canceled by a given event
`until*`	must hold up to a given event
`before*`	must hold at some time before a given event

FL termination operators are left-associative.

4.2.2.2.7 LTL operators

PSL also defines the following traditional LTL operators, each of which is equivalent to a corresponding keyword operator:

`X`	`next`
`X!`	`next!`
`F`	`eventually!`
`G`	`always`
`[ U ]`	`until!`
`[ W ]`	`until`

In each case, the LTL operator has the same precedence and associativity as its equivalent keyword operator.

4.2.2.3 Optional Branching Extension (OBE) operators

4.2.2.3.1 OBE implication operators

For any flavor of PSL, the Optional Branching Extension (OBE) operators with the 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

4.2.2.3.2 OBE occurrence operators

For any flavor of PSL, the OBE 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 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 Macros

PSL provides macro-processing capabilities that facilitate the definition of properties. All flavors support cpp-style pre-processing directives (e.g., `#define, #ifdef, #else, #include,` and `#undef`). All flavors also support special macros for %for and %if, which can be used to conditionally or iteratively generate PSL statements.

4.2.3.1 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
```

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 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 Verilog flavor, 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 EDL 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. These words appear in a larger font for distinction. 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 *VUnitType*.

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

VUnitType ::= **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

| SERE **[*** [Range] **]**

indicates that (the outer) square brackets are part of the syntax for this SERE, 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 PATH_SYM =  Verilog: . / VHDL: : / EDL: /
```

shows the path symbol macro (`PATH_SYM`).

PSL also defines a few extensions to Verilog declarations, and one extension to both Verilog and VHDL expressions, as shown in Box 1.

```
Extended_Verilog_Declaration ::=
      Verilog_module_or_generate_item_declaration
    | Extended_Verilog_Type_Declaration
Extended_Verilog_Expression ::=
      Verilog_expression
    | Verilog_Union_Expression
Extended_VHDL_Expression ::=
      VHDL_expression
    | VHDL_Union_Expression
```

Box 1—Flavor macro HDL_UNIT

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 =
    Verilog: Verilog_module_declaration / VHDL: VHDL_design_unit /
    EDL: EDL_module_declaration
```

Box 2—Flavor macro HDL_UNIT

4.3.2.2 HDL_ID and PATH_SYM

Names declared in PSL shall follow the rules for identifiers in the underlying HDL, hence, the definition of `HDL_ID` as a flavor macro. Also, pathnames shall be constructed with the separator character appropriate for the HDL, thus, the definition of `PATH_SYM`. Both of these are shown in Box 3.

Box 3—Flavor macros HDL_ID and PATH_SYM

```
Flavor Macro HDL_ID =
    Verilog: Verilog_Identifier / VHDL: VHDL_Identifier / EDL: EDL_Identifier
Flavor Macro PATH_SYM =
    Verilog: . / VHDL: : / EDL: /
```

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 =
    Verilog: Extended_Verilog_Declaration / VHDL: VHDL_declaration /
    EDL: EDL_module_item_declaration
Flavor Macro HDL_STMT =
    Verilog: Verilog_module_or_generate_item / VHDL: VHDL_concurrent_statement /
    EDL: EDL_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 =
    Verilog: Extended_Verilog_Expression / VHDL: Extended_VHDL_Expression /
    EDL: EDL_Expression
```

Box 5—Flavor macro HDL_EXPR

4.3.2.5 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 6.

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

Box 6—Flavor macros AND_OP, OR_OP, and NOT_OP

4.3.2.6 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 7.

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

Box 7— 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*.

4.3.2.7 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 8.

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

Box 8—Flavor macro LEFT_SYM and RIGHT_SYM

4.3.2.8 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 9.

```
Flavor Macro DEF_SYM  =
    Verilog: = / VHDL: is / EDL: :=
```

Box 9—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 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 which uses a non-negated strong operator is a liveness property, while one that contains only non-negated weak operators is a safety property.

4.4.4 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.

4.4.5 Simple subset

PSL can express properties which 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:

— Negation (`!`) is applied only to Booleans.
— `never` and `eventually!` are applied only to Booleans or to SEREs.
— The left-hand side of a logical `and` is Boolean.
— The left-hand side of a logical `or` is Boolean.
— The left-hand side of a logical implication (`->`) is Boolean.
— Both sides of a logical iff (`<->`) operator are Boolean.
— The right-hand side of a non-overlapping `until*` operator is Boolean.
— Both sides of an overlapping `until*` operator are Boolean.
— Both sides 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, both weak and strong suffix implication (`{} |-> {}` and `{} |-> {}!`), and any application of the `within` and `whilenot` operators to a SERE are supported in the simple subset.

4.4.6 Finite-length versus infinite-length behavior

The semantics of PSL allow us to decide whether a PSL Foundation Language 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. Consequently, liveness properties may appear not to hold just because the end of the simulation was reached, rather than because of an error in the design.

Similarly, a safety property may be satisfied on a finite-length behavior, but that does not imply that it also holds on a (possibly infinite) extension of that behavior.

5. Boolean layer

The *Boolean layer* consists of Boolean expressions, shown in Box 10, the syntax and semantics of which are dictated by the flavor of PSL being used. The Boolean layer also includes certain PSL expressions that are of Boolean type.

```
Boolean ::=
    boolean_HDL_or_PSL_Expression
```

Box 10—Boolean expression

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

5.1 HDL expressions

A Boolean HDL expression, shown in Box 11, is any HDL expression that the HDL allows to be used as the condition of an `if` statement.

```
HDL_or_PSL_Expression ::=
    HDL_Expression
HDL_Expression ::=
    HDL_EXPR
```

Box 11—Boolean HDL expression

Restrictions

In a given flavor of PSL, the value of a Boolean HDL expression is interpreted as a logical value according to the same rules that govern interpretation of that expression as the condition of an if statement in that flavor.

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.

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

- For the Verilog and EDL 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.

It is an error if more than one declaration of a given name appears in the current verification unit, or in the transitive closure of all inherited verification units, or in the default verification mode.

See 7.2 for an explanation of verification units and modes.

5.2 PSL expressions

PSL defines a collection of predefined functions that return Boolean values. These predefined functions are described in 8.1.4.

PSL also defines a special variable called an *endpoint*, which signals the completion of a sequence. Endpoint declarations and instantiations are described in 6.1.3.1 and 6.1.3.2, 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.

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* allowed by the modeling layer can be used as a clock expression.

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 EDL flavor, any expression that EDL 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 12, 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 Boolean ;
```

Box 12—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 `restrict`, `restrict_guarantee`, or `cover` 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 more than one default clock declaration appears in the current verification unit, or in the transitive closure of all inherited verification units, or 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 clk1);
cover {req; ack; !req; !ack} @(posedge clk1);
```

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 three types of building blocks:

— Boolean expressions
— sequences (which are themselves built from Boolean expressions)
— subordinate properties

Boolean 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.

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).

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 which 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 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 Sugar Extended Regular Expressions (SEREs)

Sugar Extended Regular Expressions (*SERE*s), shown in Box 13, describe single- or multi-cycle behavior built from a series of Boolean expressions.

```
SERE ::=
      Boolean
    | Sequence
Sequence ::=
    { SERE }
```

Box 13—SEREs and Sequences

The most basic SERE is a Boolean expression. A Sequence (a SERE enclosed in curly braces) is also a SERE. Both are sequential expressions.

More complex sequential expressions are built from Boolean expressions using various SERE construction and sequence composition operators. These operators are described in the sections that follow.

NOTES

1—A sequential expression is not a property on its own; it is a building block of a property.

2—SEREs are grouped using curly braces (`{}`), as opposed to Boolean expressions which are grouped using parentheses (`()`).

3—A Sequence can also be an instance of a Sequence declaration; see Sections 6.1.2.1 and 6.1.2.2.

6.1.1.1 SERE construction

6.1.1.1.1 Clocked SERE (@)

The *SERE clock* operator (**@**), shown in Box 14, provides a way to clock a SERE.

```
SERE ::=
    SERE @ clock_Boolean
```

Box 14—SERE clock operator

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

Restrictions

None.

Informal semantics

For unclocked SERE A and Boolean CLK:

> A@CLK holds on a given path *iff* (if and only if) 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 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.

NOTE—There is only one form of a clocked sere. In contrast, a distinction between weak and strong clocks is made for a clocked property (see Section 6.2.1.1).

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. 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.

6.1.1.1.2 SERE concatenation (;)

The *SERE concatenation* operator (**;**), shown in Box 15, constructs a SERE that is the concatenation of two other SEREs.

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

Box 15—SERE concatenation operator

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

Restrictions

None.

Informal semantics

For SEREs A and B:

> A;B holds 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 $n+1^{th}$ cycle.

6.1.1.1.3 Repetition operators

The repetition operators (**[]**) describe succinctly repeated concatenation of the same SERE. There are three kinds of repetition, each of which is detailed in the following subsections.

6.1.1.1.4 SERE consecutive repetition ([*])

The *SERE consecutive repetition* operator (**[*]**), shown in Box 16, constructs repeated consecutive concatenation of the same SERE.

```
SERE ::=
      SERE [ * [ Count ] ]
    | [ * [ Count ] ]
    | SERE [ + ]
    | [ + ]
Count ::=
      Number
    | Range
Range ::=
    LowBound RANGE_SYM HighBound
LowBound ::=
    Number | MIN_VAL
HighBound ::=
    Number | MAX_VAL
```

Box 16—SERE consecutive repetition operator

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

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

Otherwise, if the `Count` is a range, then the SERE describes describes any number of repetitions such that the number falls within the specified range. If the high value of the range (HighBound) is not specified (or is specified as `inf`), the SERE describes at least as many repetitions as the low value of the range. If the low value of the range (LowBound) is not specified (or is specified as `0`), the SERE describes at most as many repetitions as the high value of the range. If neither of the range values is specified, the SERE describes any number of repetitions, including zero, i.e., the empty path is also described.

When there is no SERE operand and only a Count, the resulting 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 SERE contains a Count, and the Count is a Number, then the Number shall be statically computable. If the 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 SERE `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.

6.1.1.1.5 SERE non-consecutive repetition ([=])

The *SERE non-consecutive repetition* operator (**[=]**), shown in Box 17, constructs repeated (possibly non-consecutive) concatenation of a Boolean expression.

```
SERE ::=
    Boolean [ = Count ]
Count ::=
     Number
    | Range
Range ::=
    LowBound RANGE_SYM HighBound
LowBound ::=
    Number | MIN_VAL
HighBound ::=
    Number | MAX_VAL
```

Box 17—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 SERE describes exactly that number of repetitions.

Otherwise, if the Count is a range, then the SERE describes any number of repetitions such that the number falls within the specified range. If the high value of the range (HighBound) is not specified (or is specified as `inf`), the SERE describes at least as many repetitions as the low value of the range. If the low value of the range (LowBound) is not specified (or is specified as `0`), the SERE describes at most as many repetitions as the high value of the range. If neither of the range values is specified, the SERE describes any number of repetitions, including zero, i.e., the empty path is also described.

Restrictions

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

6.1.1.1.6 SERE goto repetition ([->])

The *SERE goto repetition* operator (**[->]**), shown in Box 18, constructs repeated (possibly non-consecutive) concatenation of a Boolean expression, such that the Boolean expression holds on the last cycle of the path.

```
SERE ::=
    Boolean [ -> [ positive_Count ] ]
Count ::=
      Number
    | Range
Range ::=
    LowBound RANGE_SYM HighBound
LowBound ::=
    Number | MIN_VAL
HighBound ::=
    Number | MAX_VAL
```

Box 18—SERE goto repetition operator

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

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

Otherwise, if the Count is a range, then the SERE describes any number of repetitions such that the number falls within the specified range. If the high value of the range (HighBound) is not specified (or is specified as `inf`), the SERE describes at least as many repetitions as the low value of the range. If the low value of the range (LowBound) is not specified, the SERE describes at most as many repetitions as the high value of the range. If neither of the range values is specified, the 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 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 positive and 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.

6.1.1.2 Sequence composition

6.1.1.2.1 Sequence fusion (:)

The *sequence fusion* operator (**:**), shown in Box 19, constructs a SERE in which two sequences overlap by one cycle. That is, the second sequence starts at the cycle in which the first sequence completes.

```
SERE ::=
    Sequence : Sequence
Sequence ::=
    { SERE }
```

Box 19—Sequence fusion operator

The operands of : are both sequences, i.e., brace-enclosed SEREs.

Restrictions

None.

Informal semantics

For sequences 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.

6.1.1.2.2 Sequence or (|)

The *sequence or* operator (|), shown in Box 20, constructs a SERE in which one of two alternative sequences hold at the current cycle.

```
SERE ::=
    Sequence | Sequence
```

Box 20—Sequence or operator

The operands of | are both Sequences, i.e., brace-enclosed SEREs.

Restrictions

None.

Informal semantics

For sequences A and B:

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

6.1.1.2.3 Sequence non-length-matching and (&)

The *sequence non-length-matching and* operator (**&**), shown in Box 21, constructs a SERE in which two sequences both hold at the current cycle, regardless of whether they complete in the same cycle or in different cycles.

```
SERE ::=
    Sequence & Sequence
```

Box 21—Sequence non-length-matching and operator

The operands of & are both Sequences, i.e., brace-enclosed SEREs.

Restrictions

None.

Informal semantics

For sequences 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.

6.1.1.2.4 Sequence length-matching and (&&)

The *sequence length-matching and* operator (**&&**), shown in Box 22, constructs a SERE in which two sequences both hold at the current cycle, and furthermore both complete in the same cycle.

```
SERE ::=
    Sequence && Sequence
```

Box 22—Sequence length-matching and operator

The operands of && are both Sequences, i.e., brace-enclosed SEREs.

Restrictions

None.

Informal semantics

For sequences A and B:

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

6.1.2 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.

6.1.2.1 Sequence declaration

A *sequence declaration*, shown in Box 23, 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 Name [ ( Formal_Parameter_List ) ] DEF_SYM Sequence ;
Formal_Parameter_List ::=
    Formal_Parameter { ; Formal_Parameter }
Formal_Parameter ::=
    sequence_ParamKind Name { , Name }
sequence_ParamKind ::=
    const | boolean | sequence
```

Box 23—Sequence declaration

Restrictions

The Name of a declared sequence shall not be the same as the name of any other PSL declaration. 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.

6.1.2.2 Sequence instantiation

A *sequence instantiation*, shown in Box 24, creates an instance of a named sequence and provides actual parameters for formal parameters (if any) of the named sequence.

Sequence ::=
    *sequence*_Name [**(** Actual_Parameter_List **)**]
Actual_Parameter_List ::=
    *sequence*_Actual_Parameter { **,** *sequence*_Actual_Parameter }
*sequence*_Actual_Parameter ::=
    Number | Boolean | Sequence

Box 24—Sequence instantiation

Restrictions

For each formal parameter of the named sequence *`sequence`*`_Name`, 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 6.1.2.1,

```
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 }
```

6.1.3 Named endpoints

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

6.1.3.1 Endpoint declaration

An *endpoint declaration*, shown in Box 25, 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 Name [**(** Formal_Parameter_List **)**] DEF_SYM Sequence **;**
Formal_Parameter_List ::=
  Formal_Parameter { **;** Formal_Parameter }
Formal_Parameter ::=
  *sequence*_ParamKind Name { **,** Name }
*sequence*_ParamKind ::=
  const | **boolean** | **sequence**

Box 25—Endpoint declaration

Restrictions

The Name of an endpoint shall not be the same as the name of any other PSL declaration. 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.

6.1.3.2 Endpoint instantiation

An *endpoint instantiation*, shown in Box 26, 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 26—Endpoint instantiation

Restrictions

For each formal parameter of the named endpoint *`endpoint`*`_Name`, 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 6.1.3.1,

```
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 which are related. A family of operators usually share a common prefix, which is the name of the family, and optional suffixes which 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 27, describe single- or multi-cycle behavior built from Boolean expressions, sequential expressions, and subordinate properties.

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

Box 27—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 which are grouped using curly braces (`{}`).

6.2.1.1 Clocked FL properties

The *FL clock operator* operator (**@**), shown in Box 28, provides a way to clock an FL Property.

```
FL_Property ::=
    FL_Property @ clock_Boolean
  | FL_Property @ clock_Boolean !
```

Box 28—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, or if `CLK` never holds on that path. `A@CLK!` holds on a given path iff `CLK` holds at least once on the given path, and 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 outer clock causes alignment before the inner clock is considered. For example, `(A@CLK_A)@CLK_B` holds on a given path iff, starting at the first cycle of the given path in which `CLK_B` holds, `A` holds on the path obtained by extracting from the given path those cycles in which `CLK_A` holds.

NOTE—A distinction between weak and strong clocks is made for a clocked property. In contrast, there is only one form of a clocked SERE (see Section 6.1.1.1.1), although it is syntactically similar to the weak clocking of properties.

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.

6.2.1.2 Simple FL properties

6.2.1.2.1 always

The **always** operator, shown in Box 29, specifies that an FL property or a sequence holds at all times, starting from the present.

```
FL_Property ::=
    always FL_Property
  | always Sequence
```

Box 29—always operator

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

Restrictions

None.

Informal semantics

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

NOTE—If the operand (FL property or sequence) 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.

6.2.1.2.2 never

The **never** operator, shown in Box 30, specifies that an FL property or a sequence never holds.

```
FL_Property ::=
    never FL_Property
  | never Sequence
```

Box 30—never operator

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

Restrictions

Within the simple subset (see Section 4.4.5), the operand of a `never` property is restricted to be a Boolean expression or a SERE.

Informal semantics

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

6.2.1.2.3 eventually!

The **eventually!** operator, shown in Box 31, specifies that an FL property or a Sequence holds at the current cycle or at some future cycle.

```
FL_Property ::=
    eventually! FL_Property
  | eventually! Sequence
```

Box 31—eventually! operator

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

Restrictions

Within the simple subset (see Section 4.4.5), the operand of an `eventually!` property is restricted to be a Boolean or a SERE.

Informal semantics

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

6.2.1.2.4 next

The **next** family of operators, shown in Box 32, 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 32—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)`.

6.2.1.3 Extended next FL properties

6.2.1.3.1 next_a

The **next_a** family of operators, shown in Box 33, 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 33—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.

6.2.1.3.2 next_e

The **next_e** family of operators, shown in Box 34, 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 34—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.

6.2.1.3.3 next_event

The **next_event** family of operators, shown in Box 35, 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 35—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)`.

6.2.1.3.4 next_event_a

The **next_event_a** family of operators, shown in Box 36, 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 36—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 leftmost 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.

.

6.2.1.3.5 next_event_e

The **next_event_e** family of operators, shown in Box 37, 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 37—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.

6.2.1.4 Compound FL properties

6.2.1.4.1 abort

The **abort** operator, shown in Box 38, specifies a condition that removes any obligation for a given FL property to hold.

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

Box 38—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 which 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);
```

6.2.1.4.2 before

The **before** family of operators, shown in Box 39, 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 39—before operators

The left operand of the `before` family of operators is an FL Property that holds before the FL Property which 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 4.4.5), 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.

6.2.1.4.3 until

The **until** family of operators, shown in Box 40, 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 40—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 4.4.5), 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.

6.2.1.5 Sequence-based FL properties

6.2.1.5.1 Suffix implication

The *suffix implication* family of operators, shown in Box 41, specify that an FL property or sequence holds if some pre-requisite sequence holds.

```
FL_Property ::=
    Sequence ( FL_Property )
  | Sequence |-> Sequence !
  | Sequence |-> Sequence
  | Sequence |=> Sequence !
  | Sequence |=> Sequence
```

Box 41—Suffix implication operators

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

The `Sequence |-> Sequence!` and `Sequence |=> Sequence!` properties are strong properties, so they specify that the rightmost Sequence completes.

The `Sequence |-> Sequence` and `Sequence |=> Sequence` properties are weak properties, so they do not specify that the rightmost Sequence necessarily completes (this can happen, for example, if the rightmost Sequence contains a `[*]`).

Restrictions

None.

Informal semantics

- A `Sequence (FL_Property)` 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 any cycle C such that the left operand holds tightly from the given cycle to C.
- A `Sequence |-> Sequence!` 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 Sequence 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 |-> Sequence` 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) in any cycle C such that the Sequence that is the left operand holds tightly from the given cycle to C, either
     i) the Sequence that is the right operand holds, or
     ii) any prefix of the path beginning at C can be extended such that the Sequence that is the right operand holds tightly on the extension.
- A `Sequence |=> Sequence!` 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 Sequence that is the right operand holds in the cycle after any cycle C such that the Sequence that is the left operand holds tightly from the given cycle to C.
- A `Sequence |=> Sequence` 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) in the cycle after any cycle C such that the Sequence that is the left operand holds tightly from the given cycle to C, either
   i) the Sequence that is the right operand holds, or
   ii) any prefix of the path beginning in the cycle after C can be extended such that the Sequence that is the right operand holds tightly on the extension.

6.2.1.5.2 whilenot

The **whilenot** family of operators, shown in Box 42, specify that a sequence holds on the interval between the current cycle and a terminating condition.

```
FL_Property ::=
    whilenot! ( Boolean ) Sequence
  | whilenot ( Boolean ) Sequence
  | whilenot!_ ( Boolean ) Sequence
  | whilenot_ ( Boolean ) Sequence
```

Box 42—whilenot operators

The left operand of the `whilenot` family of operators is a Boolean expression that defines the end of the interval in which the sequence holds. The left operand is called the “terminating condition”. The right operand is the sequence that holds within the interval.

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

The `whilenot` and `whilenot_` operators are weak operators, thus they do not specify that the terminating condition eventually holds (and if the terminating condition does not eventually hold, then the sequence that is the right operand starts but never completes).

The `whilenot!` and `whilenot` operators are non-inclusive operators, that is, they specify that the sequence completes strictly before the terminating condition.

The `whilenot!_` and `whilenot_` operators are inclusive operators, that is, they specify that the sequence completes in the same cycle as the terminating condition.

Restrictions

None.

Informal semantics

— A `whilenot!` property holds in a given cycle of a given path iff either
   1) the terminating condition holds in the given cycle and the Sequence operand holds on the empty path, or
   2) there is a cycle C subsequent to the given cycle such that the terminating condition holds at C, the terminating condition does not hold at any cycle from the given cycle to the cycle before C, and the Sequence operand holds tightly from the given cycle to the cycle before C.

— A `whilenot!_` property holds in a given cycle of a given path iff either
   1) the terminating condition holds in the given cycle and the Sequence operand holds tightly from the given cycle to itself, or

2) there is a cycle C subsequent to the given cycle such that the terminating condition holds at C, the terminating condition does not hold at any cycle from the given cycle to the cycle before C, and the Sequence operand holds tightly from the given cycle to C.

— A `whilenot` property holds in a given cycle of a given path iff

1) the terminating condition holds in the given cycle and the Sequence operand holds on the empty path; or
2) there is a cycle C subsequent to the given cycle such that the terminating condition holds at C, the terminating condition does not hold at any cycle from the given cycle to the cycle before C, and the Sequence operand holds tightly from the given cycle to the cycle before C; or
3) any prefix of the given path can be extended such that the corresponding `whilenot` sequence holds tightly on the extended path and the termination condition does not hold on any cycle of the extended path.

— A `whilenot_` property holds in a given cycle of a given path iff

1) the terminating condition holds in the given cycle and the Sequence operand holds tightly from the given cycle to itself; or
2) there is a cycle C subsequent to the given cycle such that the terminating condition holds at C, the terminating condition does not hold at any cycle from the given cycle to the cycle before C, and the Sequence operand holds tightly from the given cycle to C; or
3) any prefix of the given path can be extended such that the corresponding `whilenot_` sequence holds tightly on the extended path and the terminating condition does not hold on any cycle of the extended path.

6.2.1.5.3 within

The **within** family of operators, shown in Box 43, specify that a given sequence holds on an interval starting with either the occurrence of an initial condition or completion of an initial sequence, and ending with the occurrence of a terminating condition.

```
FL_Property ::=
    within! ( Sequence_or_Boolean , Boolean ) Sequence
  | within ( Sequence_or_Boolean , Boolean ) Sequence
  | within!_ ( Sequence_or_Boolean , Boolean ) Sequence
  | within_ ( Sequence_or_Boolean , Boolean ) Sequence
Sequence_or_Boolean ::=
  Sequence | Boolean
```

Box 43—within operators

The leftmost operand of the `within` family of operators is a Sequence or Boolean expression that defines the beginning of an interval. If the leftmost operand is a Sequence, then completion of that Sequence defines the beginning of the interval. If the leftmost operand is a Boolean expression, then it is treated as if it were a Sequence containing exactly that Boolean expression. The middle operand is a Boolean condition (the “terminating condition”), the occurrence of which defines the end of the interval. The rightmost operand is a Sequence that holds on the interval.

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

The `within` and `within_` operators are weak operators, thus they do not specify that the terminating condition eventually holds (and if the terminating condition does not eventually hold, then the sequence that is the rightmost operand starts but never completes).

The `within!` and `within` operators are non-inclusive operators, that is, they specify that the sequence that is the rightmost operand completes strictly before the cycle in which the terminating condition holds.

The `within!_` and `within_` operators are inclusive operators, that is, they specify that the sequence which is the rightmost operand completes at the same cycle as that in which the terminating condition holds.

Restrictions

None.

Informal semantics

- A `within!` property holds in a given cycle of a given path iff
  1) the leftmost operand does not hold at the given cycle, or
  2) in any cycle C such that the leftmost operand holds tightly from the given cycle to C, either
     i) the terminating condition holds at C and the rightmost operand holds on the empty path, or
     ii) there is a cycle D subsequent to C such that the terminating condition holds at D, the terminating condition does not hold at any cycle from C to the cycle before D, and the rightmost operand holds tightly from C to the cycle before D.
- A `within!_` property holds in a given cycle of a given path iff
  1) the leftmost operand does not hold at the given cycle, or
  2) in any cycle C such that the leftmost operand holds tightly from the given cycle to C, either
     i) the terminating condition holds at C and the rightmost operand holds tightly from C to itself, or
     ii) there is a cycle D subsequent to C such that the terminating condition holds at D, the terminating condition does not hold at any cycle from C to the cycle before D, and the rightmost operand holds tightly from C to D.
- A `within` property holds in a given cycle of a given path iff:
  1) the leftmost operand does not hold at the given cycle, or
  2) in any cycle C such that the leftmost operand holds tightly from the given cycle to C, either
     i) the terminating condition holds at C and the rightmost operand holds on the empty path, or
     ii) there is a cycle D subsequent to C such that the terminating condition holds at D, the terminating condition does not hold at any cycle from C to the cycle before D, and the rightmost operand holds tightly from C to the cycle before D, or
     iii) any prefix of the path beginning at C can be extended such that the corresponding within sequence holds tightly on the extended path and the termination condition does not hold on any cycle of the extended path.
- A `within_` property holds in a given cycle of a given path iff:
  1) the leftmost operand does not hold at the given cycle, or
  2) in any cycle C such that the leftmost operand holds tightly from the given cycle to C, either
     i) the terminating condition holds at C and the rightmost operand holds tightly from C to itself, or
     ii) there is a cycle D subsequent to C such that the terminating condition holds at D, the terminating condition does not hold at any cycle from C to the cycle before D, and the rightmost operand holds tightly from C to D, or
     iii) any prefix of the path beginning at C can be extended such that the corresponding within_ sequence holds tightly on the extended path and the termination condition does not hold on any cycle of the extended path.

6.2.1.6 Logical FL properties

6.2.1.6.1 Logical implication

The *logical implication* operator (->), shown in Box 44, is used to specify logical implication.

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

Box 44—Logical implication operator

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

Restrictions

Within the simple subset (see Section 4.4.5), 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.

6.2.1.6.2 Logical iff

The *logical iff* operator (<->), shown in Box 45, is used to specify the iff (if and only if) relation between two properties.

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

Box 45—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.

Restrictions

Within the simple subset (see Section 4.4.5), 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.

6.2.1.6.3 Logical and

The *logical and* operator, shown in Box 46, is used to specify logical and.

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

Box 46—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 4.4.5), 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.

6.2.1.6.4 Logical or

The *logical or* operator, shown in Box 47, is used to specify logical or.

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

Box 47—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 4.4.5), 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.

6.2.1.6.5 Logical not

The *logical not* operator, shown in Box 48, is used to specify logical negation.

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

Box 48—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 4.4.5), 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.

6.2.1.7 LTL operators

The *LTL operators*, shown in Box 49, 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 49—LTL operators

The standard LTL operators are alternate syntax for the equivalent PSL operators, as shown in Table 3.

Table 3—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 50, 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 50—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 51, specifies that an OBE property holds at all next states of the given state.

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

Box 51—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 52, specifies that an OBE property holds at the given state and at all future states.

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

Box 52—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 53, 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 53—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 54, 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 54—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 55, 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 55—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 which is the operand holds at the next state.

6.2.2.2.2 EG operator

The **EG** operator, shown in Box 56, 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 56—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 57, 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 57—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 58, 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 58—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 59, is used to specify logical implication.

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

Box 59—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 60, is used to specify the *iff* (if and only if) relation between two properties.

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

Box 60—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 61, is used to specify logical and.

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

Box 61—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 62, is used to specify logical or.

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

Box 62—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 63, is used to specify logical negation.

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

Box 63—OBE not operator

The operand of the OBE not operator is an OBE Property which 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 64. The first operand of the replicated property is a `Replicator` and the second operand is a parameterized property.

```
Property ::=
    Replicator Property
Replicator ::=
    forall Name [ IndexRange ] in ValueSet :
IndexRange ::=
    LEFT_SYM finite_Range RIGHT_SYM
Flavor Macro LEFT_SYM =
    Verilog: [ / VHDL: ( / EDL: (
Flavor Macro RIGHT_SYM =
    Verilog: ] / VHDL: ) / EDL: )
ValueSet ::=
     { ValueRange { , ValueRange } }
   | boolean
ValueRange ::=
     Value
   | finite_Range
Range ::=
    LowBound RANGE_SYM HighBound
```

Box 64—Replicating properties

The first operand of a `Replicator` is the parameter in the parameterized property. This parameter can be an array. The second operand is 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 `ValueRange` specifies the set of all values within the given range.
— The comma (**,**) between *ValueRanges* indicates the union of the obtained sets.

Restrictions

If the Name has an associated IndexRange, the IndexRange 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 ValueRange, the Value shall be statically computable.

If a Range is used to specify a ValueRange, 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 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 Name.

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

Replicators can be nested, but all nested Replicators shall be at the top level. A replicated property shall not be nested within a non-replicated property.

NOTE—The Name defined by a replicator represents a non-static variable. Since the bounds of both an IndexRange and a ValueRange must be defined by statically computable expressions, those expressions cannot refer to the replication variable Name of another Replicator, and therefore neither the IndexRange nor the ValueRange of a nested Replicator can be defined in terms of the replicator variable Name of a containing Replicator.

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 65, 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 Name [ ( Formal_Parameter_List ) ] DEF_SYM Property ;
Formal_Parameter_List ::=
    Formal_Parameter { ; Formal_Parameter }
Formal_Parameter ::=
    ParamKind Name { , Name }
ParamKind ::=
    const | boolean | property | sequence
```

Box 65—Property declaration

Restrictions

The `Name` of a declared property shall not be the same as the name of any other PSL declaration.

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 66, 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 66—Property instantiation

Restrictions

For each formal parameter of the named property *property*_Name, 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* which tell the verification tools what to do with the specified properties. The verification layer also provides constructs which group related directives and other PSL statements.

7.1 Verification directives

The verification directives are:

— `assert`
— `assume`
— `assume_guarantee`
— `restrict`
— `restrict_guarantee`
— `cover`
— `fairness` and `strong fairness`

7.1.1 assert

The verification directive `assert`, shown in Box 67, instructs the verification tool to verify that a property holds.

Assert_Statement ::=
assert Property **;**

Box 67—Assert statement

Example

The directive

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

instructs the verification tool to verify that the property

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

holds in the design.

7.1.2 assume

The verification directive `assume`, shown in Box 68, instructs the verification tool to constrain the verification (e.g., the behavior of the input signals) so that a property holds.

Assume_Statement ::=
assume Property **;**

Box 68—Assume statement

Restrictions

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

Verification tools are not obligated to verify the assumed property. 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 which obey the assumption.

7.1.3 assume_guarantee

The `assume_guarantee` directive, shown in Box 69, 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_Statement ::=
assume_guarantee Property **;**

Box 69—Assume_guarantee statement

Restrictions

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

7.1.4 restrict

The verification directive `restrict`, shown in Box 70, is a way to constrain the design inputs using sequences.

```
Restrict_Statement ::=
    restrict Sequence ;
```

Box 70—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 71, instructs the verification tool to constrain the design inputs so that a sequence holds and also to verify that the restrict sequence holds.

```
Restrict_Guarantee_Statement ::=
    restrict_guarantee Sequence ;
```

Box 71—Restrict_guarantee statement

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.

7.1.6 cover

The verification directive `cover`, shown in Box 72, 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_Statement ::=
    cover Sequence ;
```

Box 72—Cover statement

Example

The directive

```
cover {start_trans;!end_trans[*];start_trans & end_trans};
```

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 which the previous one completed.

7.1.7 fairness and strong fairness

The directives `fairness` and `strong fairness`, shown in Box 73, are special kinds of assumptions which correspond to liveness properties.

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

Box 73—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 GF 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 (GF p) -> (GF q);
```

7.2 Verification units

A *verification unit*, shown in Box 74, is used to group verification directives and other PSL statements.

```
Verification_Unit ::=
    VUnitType Name [ ( Hierarchical_HDL_Name ) ] {
        { Inherit_Spec }
        { VUnit_Item }
    }
VUnitType ::=
    vunit | vprop | vmode
Name ::=
    HDL_ID
Hierarchical_HDL_Name ::=
    module_Name { PATH_SYM instance_Name }
Inherit_Spec ::=
    inherit vunit_Name { , vunit_Name } ;
VUnit_Item ::=
      HDL_Decl_or_Stmt
    | PSL_Declaration
    | Verification_Directive
```

Box 74—Verification unit

The *Name* is the name by which this verification unit is known to the verification tools.

The optional *Hierarchical_HDL_Name* indicates the design module or module instance to which the verification unit is bound. If the *Hierarchical_HDL_Name* is not present, then the verification unit binds to the top-level module of the design under verification. 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* is a verification directive or other PSL statement grouped by this verification unit. See 7.2.3 for a discussion of which PSL statements can be grouped by verification units.

The *VUnitType* 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 default Verification Unit, if it exists, shall be of type `vmode`.

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 bound to a module (as opposed to a module instance), then this is equivalent to duplicating the contents of the verification unit and binding each duplication to one instance.

Examples

```
vunit ex1a(top_block.i1.i2) {
  assert never (ena && enb);
}
```

`vunit ex1a` is bound to instance `top_block.i1.i2`. This is equivalent to the following non-bound `vunit ex1b`:

```
vunit ex1b {
  assert never (top_block.i1.i2.ena && top_block.i1.i2.enb);
}
```

As a second example, consider:

```
vunit ex2a(mod1) {
  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 non-bound `vunit ex2b`:

```
vunit ex2b {
  assert never(top_block.i1.i2.ena && top_block.i1.i2.enb);
  assert never(top_block.i1.i3.ena && top_block.i1.i3.enb);
}
```

The binding of a verification unit to a module or module instance affects all the names in the vunit.

```
vunit ex3a (top_block.i1) {
  property mutex = never (ena && enb);
  assert mutex;
}
```

`vunit ex3a` is bound to the instance `top_block.i1`. This is equivalent to the following non-bound `vunit ex3b`:

```
vunit ex3b {
  property tob_block.i1.mutex =
    never (tob_block.i1.ena && tob_block.i1.enb);
  assert tob_block.i1.mutex;
}
```

7.2.2 Verification unit inheritance

When a verification unit inherits another verification unit, the effect is as if the contents of the inherited verification unit had appeared within the inheriting verification unit, except:

a) The inherited verification unit is bound according to its own definition, and is not affected by the binding of the inheriting verification unit.
b) In the case where the inheriting verification unit and the inherited verification unit declare items with the same name (after taking into account the respective bindings), then the declaration in the inheriting verification unit takes precedence. A vunit can contain HDL declarations and PSL declarations.

For more on resolution of apparent declaration conflicts, see 7.2.4.

Examples

```
vunit ex4a(top_block.i1) {
  assert never (read_en && write_en);
}
vunit ex4b(top_block.i1.i2) {
  inherit ex4a;
  assert never (ena && enb);
}
```

`vunit ex4b` inherits `vunit ex4a`. This is equivalent to the following non-bound `vunit ex4c`:

```
vunit ex4c {
  assert never (top_block.i1.read_en && top_block.i1.write_en);
  assert never (top_block.i1.i2.ena  && top_block.i1.i2.enb);
}
```

As a second example, consider:

```
vunit ex5a(top_block.i1) {
  wire temp;
  assign temp = ack1 || ack2;
  assert always (reqa -> next temp);
}
vunit ex5b(top_block.i1) {
  inherit ex5a;
  wire temp;
  assign temp = ack1 || ack2 || ack3;
  assert always (reqb -> next temp);
}
```

`vunit 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 `vunit ex5b` is equivalent to the following non-bound `vunit ex5c`:

```
vunit ex5c {
  wire top_block.i1.temp;
  assign top_block.i1.temp =
      top_block.i1.ack1 || top_block.i1.ack2 || top_block.i1.ack3;

  assert always (top_block.i1.reqa -> next top_block.i1.temp);
  assert always (top_block.i1.reqb -> next top_block.i1.temp);
}
```

As an example of how binding and inheritance affect PSL declarations, consider:

```
vunit ex6a (top_block.i1) {
  property AckInOneCycle (boolean req, ack, clk)
     = always (req -> next ack) @ (posedge clk);
}
vunit ex6b (top_block.i1) {
  inherit ex6a;
  assert AckInOneCycle(req, ack, clk);
}
```

The `vunit ex6b` is equivalent to the following non-bound `vunit ex6c`:

```
vunit ex6c {
  property top_block.i1.AckInOneCycle (boolean req, ack, clk)
     = always (req -> next ack) @ (posedge clk);
  assert top_block.i1.AckInOneCycle(top_block.i1.req,
                                    top_block.i1.ack,
                                    top_block.i1.clk);
}
```

7.2.3 Verification unit contents

The declarations and statements that can be grouped inside a verification unit are:

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

7.2.4 Verification unit scoping rules

As discussed in 7.2.2, when an inheriting verification unit and an inherited verification unit declare items with the same name (after taking into account the respective bindings), then the declaration in the inheriting verification unit takes precedence. This general scoping rule has a specific use: it allows a verification unit to redeclare and/or give new behavior to a signal in the design under verification.

PSL recognizes four levels at which an identifier is declared. In order of increasing precedence, they are:

a) In the design.
b) In the default verification unit.
c) In an inherited verification unit.

d) In the current verification unit.

It is illegal for an identifier to be declared twice at the same level.

Example

```
vunit V {
  wire S;
  assign S = req || ack;
  sequence S = {req;ack}; // illegal - S already declared
  ...
}
```

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 three flavors, corresponding to Verilog, VHDL, and EDL. Each is described in the following sections.

8.1 The Verilog-flavored modeling layer

The Verilog flavor of the modeling layer consists of a synthesizable subset defined by IEEE P1364.1.

This subset of Verilog has also been augmented with the following:

— integer ranges
— structures
— non-determinism
— built-in functions `rose()`, `fell()`, `next()`, and `prev()`

as defined in the following subsections.

8.1.1 Integer ranges

The Verilog flavor of the modeling layer extends the Verilog data types with a finite integer type, shown in Box 75, where the range of values which the variable can take on is indicated at the declaration.

```
Extended_Verilog_Type_Declaration ::=
    integer Integer_Range list_of_variable_identifiers ;
Integer_Range ::=
    ( constant_expression : constant_expression )
```

Box 75—integer range declaration

The nonterminals list_of_variable_identifiers and constant_expression are defined in the syntax for IEEE 1364-2001 Verilog.

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.

8.1.2 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 76.

```
Extended_Verilog_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 76—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.

8.1.3 Non-determinism

The **union** operator specifies two values, shown in Box 77, either of which can be the value of the resulting expression.

```
Union_Expression ::=
   HDL_or_PSL_Expression union HDL_or_PSL_Expression
```

Box 77—Structure declaration

Example

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

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

8.1.4 Built-in functions rose(), fell(), next(), prev()

The Verilog-flavored modeling layer adds the built-in functions `rose()`, `fell()`, `prev()`, and `next()`, shown in Box 78.

```
Built_In_Function_Call ::=
    rose ( Boolean )
  | fell ( Boolean )
  | prev ( HDL_or_PSL_Expression [ , Number ] )
  | next ( Boolean )
```

Box 78—Built-in functions

8.1.4.1 rose()

The built-in function `rose()` is similar to `posedge` in Verilog. It takes a Boolean signal as argument and produces a Boolean 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 expression `b && !prev(b)`, where `b` is a Boolean signal. The function `rose(b)` can be used just like any other Boolean.

For four-valued logic, the value of `rose()` is extended in the same way that Verilog extends `posedge`.

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
```

8.1.4.2 fell()

The built-in function `fell()` is similar to `negedge` in Verilog. It takes a Boolean signal as argument and produces a Boolean that is true if the argument's value is 0 at the current cycle and 1 at the previous cycle, with respect 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 expression `!b & prev(b)`, where `b` is a Boolean signal. The function `fell(b)` can be used just like any other Boolean.

For four-valued logic, the value of `fell()` is extended in the same way that Verilog extends `negedge`.

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
```

8.1.4.3 prev()

The built-in function `prev()` takes an expression of arbitrary 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, `prev(data(0..31))` returns 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
```

8.1.4.4 next()

The built-in function `next()` gives the value of a signal of arbitrary 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 `rose()`, `fell()`, and `prev()`, 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
```

8.2 Other flavors

8.2.1 The VHDL-flavored modeling layer

The VHDL-flavored modeling layer remains undefined at this time. In the future, it can be defined as a synthesizable subset of VHDL, augmented with the same features with which the Verilog flavor is augmented. The additional features shall take on a VHDL-like syntax in the VHDL-flavored modeling layer and support VHDL-style comments.

8.2.2 The EDL-flavored modeling layer

The EDL-flavored modeling layer is defined separately; see [B2].

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. These words appear in a larger font for distinction. 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 *VUnitType*.

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

   VUnitType ::= **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

   | SERE **[*** [Range] **]**

   indicates that (the outer) square brackets are part of the syntax for this SERE, 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 which 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 PATH_SYM = Verilog: `.` / VHDL: `:` / EDL: `/`

shows the *path symbol* macro. 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 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 Verilog, VHDL, or EDL. PSL syntax also includes Flavor Macros which cause each flavor of PSL to match that of the underlying HDL for that flavor.

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

For VHDL, the PSL syntax refers to the following nonterminals in the IEEE 1076-1993 VHDL syntax:

— declaration
— concurrent_statement
— design_unit
— identifer
— expression

For EDL, the PSL syntax refers to the following nonterminals in the EDL syntax:

— module_item_declaration
— module_item
— module_declaration
— identifer
— expression

A.2.1 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

Extended_Verilog_Expression ::=
*Verilog*_expression
| *Verilog*_Union_Expression

Extended_VHDL_Expression ::=
*VHDL*_expression
| *VHDL*_Union_Expression

Union_Expression ::=
HDL_or_PSL_Expression **union** HDL_or_PSL_Expression

A.2.2 Flavor macros

Flavor Macro PATH_SYM =
Verilog: **.** / VHDL: **:** / EDL: **/**
Flavor Macro HDL_ID =
Verilog: *Verilog*_Identifier / VHDL: *VHDL*_Identifier / EDL: *EDL*_Identifier
Flavor Macro DEF_SYM =
Verilog: **=** / VHDL: **is** / EDL: **:=**
Flavor Macro RANGE_SYM =
Verilog: **:** / VHDL: **to** / EDL: **..**
Flavor Macro AND_OP =
Verilog: **&&** / VHDL: **and** / EDL: **&**
Flavor Macro OR_OP =
Verilog: **||** / VHDL: **or** / EDL: **|**
Flavor Macro NOT_OP =
Verilog: **!** / VHDL: **not** / EDL: **!**
Flavor Macro MIN_VAL =
Verilog: **0** / VHDL: **0** / EDL: *null*
Flavor Macro MAX_VAL =

Verilog: **inf** / VHDL: **inf** / EDL: *null*

Flavor Macro HDL_EXPR =
Verilog: Extended_Verilog_Expression / VHDL: Extended_VHDL_Expression
/ EDL: *EDL*_Expression

Flavor Macro HDL_UNIT =
Verilog: *Verilog*_module_declaration / VHDL: *VHDL*_design_unit / EDL: *EDL*_module_declaration

Flavor Macro HDL_DECL =
Verilog: Extended_Verilog_Declaration / VHDL: *VHDL*_declaration
/ EDL: *EDL*_module_item_declaration

Flavor Macro HDL_STMT =
Verilog: *Verilog*_module_or_generate_item / VHDL: *VHDL*_concurrent_statement
/ EDL: *EDL*_module_item

Flavor Macro LEFT_SYM =
Verilog: **[** / VHDL: **(** / EDL: **(**

Flavor Macro RIGHT_SYM =
Verilog: **]** / VHDL: **)** / EDL: **)**

A.3 Syntax productions

The rest of this section defines the PSL syntax.

A.3.1 Verification units

PSL_Specification ::=
{ Verification_Item }

Verification_Item ::=
HDL_UNIT | Verification_Unit

Verification_Unit ::=
VUnitType Name [**(** Hierarchical_HDL_Name **)**] **{**
{ Inherit_Spec }
{ VUnit_Item }
}

VUnitType ::=
vunit | **vprop** | **vmode**

Name ::=
HDL_ID

Hierarchical_HDL_Name ::=
*module*_Name { PATH_SYM *instance*_Name }

Inherit_Spec ::=
inherit *vunit*_Name { **,** *vunit*_Name } **;**

VUnit_Item ::=
HDL_Decl_or_Stmt
| PSL_Declaration (see A.3.2)
| Verification_Directive (see A.3.3)

HDL_Decl_or_Stmt ::=
HDL_DECL | HDL_STMT

A.3.2 PSL declarations

PSL_Declaration ::=
Property_Declaration

| Sequence_Declaration
| Endpoint_Declaration
| Clock_Declaration

Property_Declaration ::=
property Name [**(** Formal_Parameter_List **)**] DEF_SYM Property **;**

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

Formal_Parameter ::=
ParamKind Name { **,** Name }

ParamKind ::=
const | **boolean** | **property** | **sequence**

Sequence_Declaration ::=
sequence Name [**(** Formal_Parameter_List **)**] DEF_SYM Sequence **;** (see A.3.5)

Endpoint_Declaration ::=
endpoint Name [**(** Formal_Parameter_List **)**] DEF_SYM Sequence **;** (see A.3.5)

Clock_Declaration ::=
default clock DEF_SYM Boolean **;** (see A.3.7)

Actual_Parameter_List ::=
Actual_Parameter { **,** Actual_Parameter }

Actual_Parameter ::=
Number | Boolean | Property | Sequence (see A.3.7) (see A.3.7) (see A.3.4) (see A.3.5)

A.3.3 PSL statements

Verification_Directive ::=
Assert_Statement
| Assume_Statement
| Assume_Guarantee_Statement
| Restrict_Statement
| Restrict_Guarantee_Statement
| Cover_Statement
| Fairness_Statement

Assert_Statement ::=
assert Property **;** (see A.3.4)

Assume_Statement ::=
assume Property **;** (see A.3.4)

Assume_Guarantee_Statement ::=
assume_guarantee Property **;** (see A.3.4)

Restrict_Statement ::=
restrict Sequence **;** (see A.3.5)

Restrict_Guarantee_Statement ::=
restrict_guarantee Sequence **;** (see A.3.5)

Cover_Statement ::=
cover Sequence **;** (see A.3.5)

Fairness_Statement ::=
fairness Boolean **;**
| **strong fairness** Boolean **,** Boolean **;** (see A.3.7)

A.3.4 PSL properties

```
Property ::=
        Replicator Property
      | FL_Property
      | OBE_Property
Replicator ::=
      forall Name [ IndexRange ] in ValueSet :
IndexRange ::=
      LEFT_SYM finite_Range RIGHT_SYM
ValueSet ::=
       { ValueRange { , ValueRange } }
      | boolean
ValueRange ::=
       Value                                              (see A.3.7)
      | finite_Range                                      (see A.3.5)
FL_Property ::=
       Boolean                                            (see A.3.7)
      | ( FL_Property )
      | property_Name [ ( Actual_Parameter_List ) ]

      | FL_Property @ clock_Boolean [ ! ]
      | 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.3.7)
| **X [** Number **] (** FL_Property **)**
| **X! [** Number **] (** FL_Property **)**
| **next [** Number **] (** FL_Property **)**
| **next! [** Number **] (** FL_Property **)**
: (see A.3.5)
| **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.3.5)
| Sequence **(** FL_Property **)**
| Sequence **|->** Sequence [**!**]
| Sequence **|=>** Sequence [**!**]
:
| **always** Sequence
| **never** Sequence
| **eventually!** Sequence
:
| **within! (** Sequence_or_Boolean **,** Boolean **)** Sequence
| **within (** Sequence_or_Boolean **,** Boolean **)** Sequence
| **within!_ (** Sequence_or_Boolean **,** Boolean **)** Sequence
| **within_ (** Sequence_or_Boolean **,** Boolean **)** Sequence
:
| **whilenot! (** Boolean **)** Sequence
| **whilenot (** Boolean **)** Sequence
| **whilenot!_ (** Boolean **)** Sequence
| **whilenot_ (** Boolean **)** Sequence

Sequence_or_Boolean ::=
Sequence | Boolean

A.3.5 Sequences

Sequence ::=
{ SERE **}**
| *sequence*_Name [**(** Actual_Parameter_List **)**]

A.3.6 Sugar extended regular expressions

```
SERE ::=
        Boolean                                              (see A.3.7)
      | Sequence
      | SERE @ clock_Boolean
: Composition Operators :
      | SERE ; SERE
      | Sequence : Sequence
      | Sequence AndOrOp Sequence
: RegExp Qualifiers :
      | SERE [ * [ Count ] ]
      | [ * [ Count ] ]
      | SERE [ + ]
      | [ + ]
      :
      | Boolean [ = Count ]
      | Boolean [ -> [ positive_Count ] ]
AndOrOp ::=
      && | & | |

Count ::=
      Number | Range
Range ::=
      LowBound RANGE_SYM HighBound
LowBound ::=
      Number | MIN_VAL
HighBound ::=
      Number | MAX_VAL
```

A.3.7 Forms of expression

```
Value ::=
      Boolean | Number
Boolean ::=
      boolean_HDL_or_PSL_Expression
HDL_or_PSL_Expression ::=
        HDL_Expression
      | endpoint_Name [ ( Actual_Parameter_List ) ]
      | Built_In_Function_Call
      | HDL_or_PSL_Expression union HDL_or_PSL_Expression
HDL_Expression ::=
      HDL_EXPR
Built_In_Function_Call ::=
       rose ( Boolean )
      | fell ( Boolean )
      | prev ( HDL_or_PSL_Expression [ , Number ] )
      | next ( Boolean )
Number ::=
      integer_HDL_Expression
```

A.3.8 Optional branching extension

OBE_Property ::=
Boolean
| **(** OBE_Property **)**
| *property*_Name [**(** Actual_Parameter_List **)**]
: Logical Operators :
| **!** OBE_Property
| OBE_Property **&** OBE_Property
| OBE_Property **|** 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

(normative)

Formal syntax and semantics of the temporal layer

This appendix formally describes the syntax and semantics of the temporal layer.

B.1 Syntax

Boolean expression syntax varies according to the Sugar flavor used. The formal syntax definition uses the complete set $\{\neg, \wedge\}$, and semantics are given here only to these two operators. Semantics of any other boolean expression follow directly from these.

Definition 1 (Boolean expression).

- *Every atomic proposition is a boolean expression.*
- *If b, b_1, and b_2 are boolean expressions, then so are the following:*
  - (b)
  - $\neg b$
  - $b_1 \wedge b_2$

Definition 2 (Sugar Extended Regular Expressions (SEREs)).

- *Every boolean expression is a SERE.*
- *If r, r_1, and r_2 are SEREs, and clk is a boolean expression, then the following are SEREs:*
  - $\{r\}$
  - $r_1 \ ; \ r_2$
  - $r_1 \ : \ r_2$
  - $\{r_1\} \mid \{r_2\}$
  - $\{r_1\} \ \&\& \ \{r_2\}$
  - $r[*]$
  - $r@clk$

Definition 3 (Formulas of the Sugar Foundation Language (FL)).

- *Every boolean expression is a Sugar FL formula.*
- *If b and clk are boolean expressions, f, f_1, and f_2 are Sugar FL formulas and r, r_1, and r_2 are SEREs, then the following are Sugar FL formulas:*
  - (f)
  - $\neg f$
  - $f_1 \wedge f_2$
  - $X! \ f$
  - $[f_1 \ U \ f_2]$
  - $\{r\}(f)$

- $\{r_1\} \mapsto \{r_2\}!$
- $\{r_1\} \mapsto \{r_2\}$
- $f \ abort \ b$
- $f@clk$
- $f@clk!$

In Section B.3, we show additional operators which provide syntactic sugaring to those described above.

Definition 4 (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)
  - $\neg f$
  - $f_1 \wedge f_2$
  - EXf
  - $E[f_1 \ U \ f_2]$
  - EGf

Additional OBE operators are derived from these as follows [1]:

- $f_1 \vee f_2 = \neg(\neg f_1 \wedge \neg f_2)$
- $f_1 \rightarrow f_2 = \neg f_1 \vee f_2$
- $f_1 \leftrightarrow f_2 = (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$
- $EFf = E[\mathrm{T} \ U \ f]$
- $AXf = \neg EX \neg f$
- $A[f_1 \ U \ f_2] = \neg(E[\neg f_2 \ U \ (\neg f_1 \wedge \neg f_2)] \vee EG \neg f_2)$
- $AGf = \neg E[\mathrm{T} \ U \ \neg f]$
- $AFf = A[\mathrm{T} \ U \ f]$

Definition 5 (Sugar Formulas).

- *Every Sugar FL formula is a Sugar formula.*
- *Every OBE formula is a Sugar formula.*

B.2 Semantics

The semantics of a Sugar formula are defined with respect to a *model* M. 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.

[1] Where $\mathrm{T} = p \vee \neg p$ for some $p \in P$.

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.

We will denote a letter from 2^P by ℓ, and a finite or infinite word from 2^P by ω. We denote the length of word ω as $|\omega|$. A finite word $\omega = \ell_0\ell_1\ell_2\cdots\ell_n$ has length $n+1$, while an infinite word has length ∞. We denote by ω^i the suffix of ω starting at ℓ_i. That is, $\omega^i = \ell_i\ell_{i+1}\cdots\ell_n$ (or $\omega^i = \ell_i\ell_{i+1}\cdots$). We denote by $\omega^{i,j}$ the finite sequence of letters starting from ℓ_i and ending in ℓ_j. That is, $\omega^{i,j} = \ell_i\ell_{i+1}\ldots\ell_j$.

For readability, we first define the semantics of unclocked Sugar formulas (and SEREs) and only then the semantics of clocked Sugar formulas (and clocked SEREs). In fact, the semantics of unclocked Sugar formulas (and unclocked SEREs) can be obtained from the semantics of clocked Sugar formulas (and clocked SEREs) by replacing the clock context with T[2].

B.2.1 Semantics of Boolean expressions

We define the semantics of boolean expressions over letters from the alphabet 2^P, thus a letter is a subset of the set of atomic propositions P. The notation $\ell \models b$ means that boolean expression b holds under the truth assignment represented by ℓ. The semantics of boolean expressions are defined as follows, where p denotes an atomic proposition and b, b_1, and b_2 denote boolean expressions.

- $\ell \models p \iff p \in \ell$
- $\ell \models (b) \iff \ell \models b$
- $\ell \models \neg b \iff \ell \not\models b$
- $\ell \models b_1 \wedge b_2 \iff \ell \models b_1$ and $\ell \models b_2$

B.2.2 Unclocked semantics

B.2.2.1 Semantics of unclocked SEREs

The semantics of unclocked SEREs are defined over finite words from the alphabet 2^P. We will denote a finite word over 2^P by w. The concatenation of w_1 and w_2 is denoted by w_1w_2. The empty word is denoted by ϵ, so that $w\epsilon = \epsilon w = w$. The notation $w \models r$, where r is a SERE, means that w is in the language of r. The semantics of SEREs are defined as follows, where b denotes a boolean expression, r, r_1, and r_2 denote unclocked SEREs, and $[i..k]$ denotes the set of integers $\{j \ : \ i \leq j \ \wedge \ j \leq k\}$.

[2] Where $\mathrm{T} = p \vee \neg p$ for some $p \in P$.

- $w \models b \iff |w| = 1$ and $\ell_0 \models b$
- $w \models \{r\} \iff w \models r$
- $w \models r_1;r_2 \iff$ there exist w_1 and w_2 such that $w = w_1 w_2$, $w_1 \models r_1$, and $w_2 \models r_2$
- $w \models r_1:r_2 \iff$ there exist w_1, w_2, and ℓ such that $w = w_1 \ell w_2$, $w_1 \ell \models r_1$, and $\ell w_2 \models r_2$
- $w \models \{r_1\}|\{r_2\} \iff w \models r_1$ or $w \models r_2$
- $w \models \{r_1\}\,\&\&\,\{r_2\} \iff w \models r_1$ and $w \models r_2$
- $w \models r[*] \iff$ either $w = \epsilon$ or there exist $w_1, w_2, \ldots, w_j$ such that $w = w_1 w_2 \ldots w_j$ and for every $i \in [1..j]$, $w_i \models r$

B.2.2.2 Semantics of unclocked Sugar FL formulas

The semantics of Sugar FL formulas are defined over finite or infinite words from the alphabet 2^P. The notation $\omega \models f$ means that formula f holds along the (finite or infinite) word ω. The notation $M \models f$ means that $\hat{L}(\pi) \models f$ for every computation path π in M such that $\pi_0 \in S_0$. The semantics of an FL formula are defined as follows[3],where b denotes a boolean expression, r, r_1, and r_2 denote SEREs, f, f_1, and f_2 denote FL formulas, and $[i..k)$ denotes the set of integers $\{j \;:\; i \leq j \;\wedge\; j < k\}$.

- $\omega \models b \iff \ell_0 \models b$
- $\omega \models (f) \iff \omega \models f$
- $\omega \models \neg f \iff \omega \not\models f$
- $\omega \models f_1 \wedge f_2 \iff \omega \models f_1$ and $\omega \models f_2$
- $\omega \models X!\, f \iff |\omega| > 1$ and $\omega^1 \models f$
- $\omega \models [f_1 \;U\; f_2] \iff$ there exists $k \in [0..|\omega|)$ such that $\omega^k \models f_2$, and for every $j \in [0..k)$, $\omega^j \models f_1$
- $\omega \models \{r\}(f) \iff$ for every $j \in [0..|\omega|)$ such that $\omega^{0,j} \models r$, $\omega^j \models f$
- $\omega \models \{r_1\} \mapsto \{r_2\}! \iff$ for every $j \in [0..|\omega|)$ such that $\omega^{0,j} \models r_1$ there exists $k \in [j..|\omega|)$ such that $\omega^{j,k} \models r_2$
- $\omega \models \{r_1\} \mapsto \{r_2\} \iff$ for every $j \in [0..|\omega|)$ such that $\omega^{0,j} \models r_1$ either there exists $k \in [j..|\omega|)$ such that $\omega^{j,k} \models r_2$ or for every $k \in [j..|\omega|)$ there exists a finite word ω' such that $\omega^{j,k}\omega' \models r_2$
- $\omega \models f$ abort $b \iff$ either $\omega \models f$ or $\omega \models b$ or there exists $j \in [1..|\omega|)$ and word ω' such that $\omega^j \models b$ and $\omega^{0,j-1}\omega' \models f$

B.2.3 Clocked semantics

In the above we disregarded the *clock operator* (@) in the definition of Sugar formulas (and SEREs). The semantics of clocked SEREs and clocked Sugar formulas are defined formally below[4].

[3] The semantics presented here for the LTL operators are the standard ones.

[4] An equivalent definition in terms of rewrite rules is given in Appendix B.5.

B.2.3.1 Semantics of clocked SEREs

Clocked SEREs are defined over finite words from the alphabet 2^P and a boolean expression that serves as the clock context. The notation $w \models^c r$, where r is a SERE and c is a boolean expression, means that w is in the language of 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, and $[i..k)$ denotes the set of integers $\{j \ : \ i \leq j \ \wedge \ j < k\}$.

- $w \models^c b \iff |w| \geq 1$, for every $i \in [0..|w|-1)$, $\ell_i \models \neg c$ and $\ell_{|w|-1} \models c \wedge b$
- $w \models^c \{r\} \iff w \models^c r$
- $w \models^c r_1; r_2 \iff$ there exists w_1 and w_2 such that $w = w_1 w_2$, $w_1 \models^c r_1$, and $w_2 \models^c r_2$
- $w \models^c r_1 : r_2 \iff$ there exists w_1, w_2, and ℓ such that $w = w_1 \ell w_2$, $w_1 \ell \models^c r_1$, and $\ell w_2 \models^c r_2$
- $w \models^c \{r_1\}|\{r_2\} \iff w \models^c r_1$ or $w \models^c r_2$
- $w \models^c \{r_1\} \,\&\&\, \{r_2\} \iff w \models^c r_1$ and $w \models^c r_2$
- $w \models^c r[*] \iff$ either $w = \epsilon$ or there exists $w_1, w_2, \ldots, w_j$ such that $w = w_1 w_2 \ldots w_j$ and for every $i \in [1..j)$, $\ w_i \models^c r$
- $w \models^c r@c_1 \iff$ there exists $i \in [0..|\omega|)$ such that $\omega^{0,i} \models^{\mathrm{T}} \{\neg c_1[*]; c_1\}$ and $\omega^i \models^{c_1} r$

B.2.3.2 Semantics of clocked Sugar FL formulas

We now turn to the semantics of clocked Sugar FL formulas. The notation $\omega \models^c f$ where f is a formula and c is a boolean expression means that formula f holds along the (finite or infinite) word ω in the context of clock c. The notation $M \models f$ means that $\hat{L}(\pi) \models^{\mathrm{T}} f$ for every computation path π in M such that $\pi_0 \in S_0$ (where $\mathrm{T} = p \vee \neg p$ for some $p \in P$). The semantics of a (clocked) Sugar FL formula are defined as follows[5], where b, c, and c_1 denote boolean expressions, r, r_1, and r_2 denote SEREs, f, f_1, and f_2 denote (clocked) FL formulas, $[i..k)$ denotes the set of integers $\{j \ : \ i \leq j \ \wedge \ j < k\}$, and $(i..k)$ denotes the set of integers $\{j \ : \ i < j \ \wedge \ j < k\}$.

- $\omega \models^c b \iff \ell_0 \models b$
- $\omega \models^c (f) \iff \omega \models^c f$
- $\omega \models^c \neg f \iff \omega \not\models^c f$
- $\omega \models^c f_1 \wedge f_2 \iff \omega \models^c f_1$ and $\omega \models^c f_2$
- $\omega \models^c X!\ f \iff$ there exists $i \in [1..|w|)$ such that $\omega^{1,i} \models^{\mathrm{T}} \{\neg c[*]; c\}$ and $\omega^i \models^c f$
- $\omega \models^c [f_1 \ U \ f_2] \iff$ there exists $k \in [0..|\omega|)$ such that $\omega^k \models^{\mathrm{T}} c$, $\omega^k \models^c f_2$, and for every $j \in [0..k)$ for which $\omega^j \models^{\mathrm{T}} c$, $\ \omega^j \models^c f_1$
- $\omega \models^c \{r\}(f) \iff$ for every $i \in [0..|\omega|)$ such that $\omega^{0,i} \models^c r$, there exists $j \in [i..|\omega|)$ such that $\omega^{i,j} \models^{\mathrm{T}} \{\neg c[*]; c\}$ and $\omega^j \models^c f$

[5] When the context is T, the semantics reduce to the unclocked semantics as previously presented. Thus, the semantics of the LTL operators in context T reduce to the standard ones.

- $\omega \models^{c} \{r_1\} \mapsto \{r_2\}! \iff$ for every $i \in [0..|\omega|)$ such that $\omega^{0,i} \models^{c} r_1$ there exists $j \in [i..|\omega|)$ such that $\omega^{i,j} \models^{c} r_2$
- $\omega \models^{c} \{r_1\} \mapsto \{r_2\} \iff$ for every $i \in [0..|\omega|)$ such that $\omega^{0,i} \models^{c} r_1$, either there exists $j \in [i..|\omega|)$ such that $\omega^{i,j} \models^{c} r_2$ or for every $j \in [i..|\omega|)$ there exists a finite word ω' such that $\omega^{i,j}\omega' \models^{c} r_2$
- $\omega \models^{c} f$ abort $b \iff$ either $\omega \models^{c} f$ or $\omega \models^{c} b$ or there exists $i \in [1..|\omega|)$ and word ω' such that $\omega^{i} \models^{\mathrm{T}} c \wedge b$ and $\omega^{0,i-1}\omega' \models^{c} f$
- $\omega \models^{c} f@c_1! \iff$ there exists $i \in [0..|\omega|)$ such that $\omega^{0,i} \models^{\mathrm{T}} \{\neg c_1[*]; c_1\}$ and $\omega^{i} \models^{c_1} f$

B.2.4 Semantics of OBE formulas

The semantics of OBE formulas are defined over states in the model, rather than finite or infinite words. 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[6], where b denotes a boolean expression and f, f_1, and f_2 denote OBE formulas.

- $M, s \models b \iff s \models b$
- $M, s \models (f) \iff M, s \models f$
- $M, s \models \neg f \iff M, s \not\models f$
- $M, s \models f_1 \wedge f_2 \iff M, s \models f_1$ 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 $M, \pi_1 \models f$
- $M, s \models E[f_1\ U\ f_2] \iff$ 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.3 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 Sugar FL operators presented above. The syntactic sugar is defined below[7].

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.

[6] The semantics are those of standard CTL.

[7] Where $\mathrm{T} = p \vee \neg p$ for some $p \in P$ and $\mathrm{F} = p \wedge \neg p$ for some $p \in P$.

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.

- $\{r_1\}$ & $\{r_2\} = \{\{r_1\}$ && $\{r_2; \mathrm{T}[*]\}\} \mid \{\{r_1; \mathrm{T}[*]\}$ && $\{r_2\}\}$
- $r[+] = r; r[*]$
- $r[*i] = \begin{cases} \mathrm{F}[*] & \text{if } i = 0 \\ \overbrace{r; r; ...; r}^{i \text{ times}} & \text{otherwise} \end{cases}$
- $r[*i..j] = \{r[*i]\}|...|\{r[*j]\}$
- $r[*i..] = \{r[*i]\}; \{r[*]\}$
- $r[*..i] = \{r[*0]\}|...|\{r[*i]\}$
- $r[*..] = r[*0..]$
- $[+] = \mathrm{T}[+]$
- $[*] = \mathrm{T}[*]$
- $[*i] = \mathrm{T}[*i]$
- $[*i..j] = \mathrm{T}[*i..j]$
- $[*i..] = \mathrm{T}[*i..]$
- $[*..i] = \mathrm{T}[*..i]$
- $[*..] = \mathrm{T}[*..]$
- $b[= i] = \{\neg b[*]; b\}[*i]; \neg b[*]$
- $b[= i..j] = \{b[= i]\}|...|\{b[= j]\}$
- $b[= i..] = b[= i]; [*]$
- $b[= ..i] = \{b[= 0]\}|...|\{b[= i]\}$
- $b[= ..] = b[= 0..]$
- $b[\rightarrow] = \neg b[*]; b$
- $b[\rightarrow k] = \{\neg b[*]; b\}[*k]$
- $b[\rightarrow k..l] = \{b[\rightarrow k]\}|...|\{b[\rightarrow l]\}$
- $b[\rightarrow k..] = \{b[\rightarrow k]\}|\{b[\rightarrow k]; [*]; b\}$
- $b[\rightarrow ..k] = \{b[\rightarrow 1]\}|...|\{b[\rightarrow k]\}$
- $b[\rightarrow ..] = b[\rightarrow 1..]$

B.3.2 Additional 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 f, f_1, and f_2 denote FL formulas.

- $f_1 \vee f_2 = \neg(\neg f_1 \wedge \neg f_2)$
- $f_1 \rightarrow f_2 = \neg f_1 \vee f_2$
- $f_1 \leftrightarrow f_2 = (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$

- $Gf = \neg F \neg f$
- $Xf = \neg X! \neg f$
- $Ff = [\mathrm{T}\ U\ f]$
- $[f_1\ W\ f_2] = [f_1\ U\ f_2] \vee Gf_1$

- $always\ f = G\ f$
- $never\ f = G\ \neg f$
- $next!\ f = X!\ f$
- $next\ f = X\ f$
- $eventually!\ f = Ff$
- $f_1\ releases\ f_2 = [f_1\ V\ f_2]$

- $f_1\ until!\ f_2 = [f_1\ U\ f_2]$
- $f_1\ until\ f_2 = [f_1\ W\ f_2]$
- $f_1\ until!\_\ f_2 = [f_1\ U\ f_1 \wedge f_2]$
- $f_1\ until\_\ f_2 = [f_1\ W\ f_1 \wedge f_2]$

- $f_1\ before!\ f_2 = [\neg f_2\ U\ f_1 \wedge \neg f_2]$
- $f_1\ before\ f_2 = [\neg f_2\ W\ f_1 \wedge \neg f_2]$
- $f_1\ before!\_\ f_2 = [\neg f_2\ U\ f_1]$
- $f_1\ before\_\ f_2 = [\neg f_2\ W\ f_1]$

- $X!\ [i]f = \overbrace{X!\ X!\ ...X!}^{i\ times}\ f$
- $X[i]f = \overbrace{XX...X}^{i\ times} f$
- $next![i]\ f = X!\ [i]\ f$
- $next[i]\ f = X[i]\ f$
- $next\_a![i..j]f = (X![i]f) \wedge \ldots \wedge (X![j]f)$
- $next\_a[i..j]f = (X[i]f) \wedge \ldots \wedge (X[j]f)$
- $next\_e![i..j]f = (X![i]f) \vee \ldots \vee (X![j]f)$
- $next\_e[i..j]f = (X[i]f) \vee \ldots \vee (X[j]f)$

- $next\_event!(b)(f) = [\neg b\ U\ b \wedge f]$
- $next\_event(b)(f) = [\neg b\ W\ b \wedge f]$
- $next\_event!(b)[k](f) = next\_event!(b)\ \overbrace{(X!\ next\_event!(b)...(X!\ next\_event!(\overbrace{b}^{k-1\ times})(f))...)}^{k-1\ times}$
- $next\_event(b)[k](f) = next\_event(b)\ \overbrace{(X next\_event(b)...(X next\_event(\overbrace{b}^{k-1\ times})(f))...)}^{k-1\ times}$
- $next\_event\_a!(b)[k..l](f) = next\_event!(b)[k](f) \wedge ... \wedge next\_event!(b)[l](f)$
- $next\_event\_a(b)[k..l](f) = next\_event(b)[k](f) \wedge ... \wedge next\_event(b)[l](f)$
- $next\_event\_e!(b)[k..l](f) = next\_event!(b)[k](f) \vee ... \vee next\_event!(b)[l](f)$
- $next\_event\_e(b)[k..l](f) = next\_event(b)[k](f) \vee ... \vee next\_event(b)[l](f)$

- $\{r_1\} \Mapsto \{r_2\}! = \{r_1\} \mapsto \{\mathrm{T}; r_2\}!$
- $\{r_1\} \Mapsto \{r_2\} = \{r_1\} \mapsto \{\mathrm{T}; r_2\}$
- $always\{r\} = \{\mathrm{T}[*]\} \mapsto \{r\}$
- $never\{r\} = \{\mathrm{T}[*]; r\} \mapsto \{\mathrm{F}\}$
- $eventually!\ \{r\} = \{\mathrm{T}\} \mapsto \{\mathrm{T}[*]; r\}!$
- $within!(r_1, b)\{r_2\} = \{r_1\} \mapsto \{r_2 \,\&\&\, b[=0]; b\}!$
- $within(r_1, b)\{r_2\} = \{r_1\} \mapsto \{r_2 \,\&\&\, b[=0]; b\}$
- $within!\_(r_1, b)\{r_2\} = \{r_1\} \mapsto \{r_2 \,\&\&\, \{b[=0]; b\}\}!$
- $within\_(r_1, b)\{r_2\} = \{r_1\} \mapsto \{r_2 \,\&\&\, \{b[=0]; b\}\}$
- $whilenot!(b)\{r\} = within!(\mathrm{T}, b)\{r\}$
- $whilenot(b)\{r\} = within(\mathrm{T}, b)\{r\}$
- $whilenot!\_(b)\{r\} = within!\_(\mathrm{T}, b)\{r\}$
- $whilenot\_(b)\{r\} = within\_(\mathrm{T}, b)\{r\}$
- $f@c = \neg(\neg f@c!)$

B.3.3 forall

If f is a Sugar formula, $v_0, v_1, \cdots, v_n$ are constants, and j, k, l and m are integers, then the following are Sugar 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:

- $forall\ i\ in\ \{v_0, v_1, \cdots, v_n\} : f = \bigwedge_{u \in \{v_0, v_1, \cdots, v_n\}} f[i \leftarrow u]$
- $forall\ i\ in\ j..k : f = \bigwedge_{u=j}^{k} f[i \leftarrow u]$
- $forall\ i\ in\ boolean : f = \bigwedge_{u=0}^{1} f[i \leftarrow u]$
- $forall\ i\langle l..m\rangle\ in\ \{v_0, v_1, \cdots, v_n\} : f = \bigwedge_{u_l \in \{v_0, v_1, \cdots, v_n\}} \cdots \bigwedge_{u_m \in \{v_0, v_1, \cdots, v_n\}} f[i\langle l..m\rangle \leftarrow \langle u_l..u_m\rangle]$
- $forall\ i\langle l..m\rangle\ in\ j..k : f = \bigwedge_{u_l=j}^{k} \cdots \bigwedge_{u_m=j}^{k} f[i\langle l..m\rangle \leftarrow \langle u_l..u_m\rangle]$
- $forall\ i\langle l..m\rangle\ in\ boolean : f = \bigwedge_{u_l=0}^{1} \cdots \bigwedge_{u_m=0}^{1} f[i\langle l..m\rangle \leftarrow \langle u_l..u_m\rangle]$

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 index j (where $l \leq j \leq m$) in the vector i by u_j.

B.4 Typed-text representation of symbols

Table 1 shows the mapping of various symbols used in this definition to the corresponding typed-text Sugar representation.

Table 1. Typed-text symbols in the Verilog, VHDL, and EDL flavors

	Verilog	VHDL	EDL
$\mapsto$	`|->`	`|->`	`|->`
$\Rightarrow$	`|=>`	`|=>`	`|=>`
$\rightarrow$	`->`	`->`	`->`
$\leftrightarrow$	`<->`	`<->`	`<->`
$\neg$	`!`	`not`	`!`
$\wedge$	`&&`	`and`	`&`
$\vee$	`||`	`or`	`|`
..	`:`	`to`	`..`
$\langle\,\rangle$	`[ ]`	`( )`	`( )`

B.5 Rewriting rules for clocks

In Section B.2 we gave the semantics of clocked Sugar formulas directly. There is an equivalent definition in terms of unclocked Sugar formulas, as follows: Starting from the outermost clock, use the following rules to translate clocked SEREs into unclocked SEREs, and clocked Sugar formulas into unclocked Sugar formulas. The rewrite rules for SEREs are:

1. $\mathcal{T}^c(b) = \{\neg c[*]; c \wedge b\}$
2. $\mathcal{T}^c(r_1 \,;\, r_2) = \mathcal{T}^c(r_1) \,;\mathcal{T}^c(r_2)$
3. $\mathcal{T}^c(r_1 : r_2) = \mathcal{T}^c(r_1) : \mathcal{T}^c(r_2)$
4. $\mathcal{T}^c(r_1 \mid r_2) = \mathcal{T}^c(r_1) \mid \mathcal{T}^c(r_2)$
5. $\mathcal{T}^c(r_1 \;\&\&\; r_2) = \mathcal{T}^c(r_1) \;\&\&\; \mathcal{T}^c(r_2)$
6. $\mathcal{T}^c(r[*]) = \{\mathcal{T}^c(r)\}[*]$
7. $\mathcal{T}^c(r@c_1) = \{\neg c_1[*];\, \{c_1{:}\mathcal{T}^{c_1}(r)\}\}$

The rewriting rules for Sugar formulas are:

1. $\mathcal{T}^c(b) = b$
2. $\mathcal{T}^c(\neg f) = \neg\mathcal{T}^c(f)$
3. $\mathcal{T}^c(f_1 \wedge f_2) = (\mathcal{T}^c(f_1) \wedge \mathcal{T}^c(f_2))$

4. $\mathcal{T}^c(\mathsf{X!}\, f) = \mathsf{X!}\, [\neg c \;\mathsf{U}\; (c \wedge \mathcal{T}^c(f))]$
5. $\mathcal{T}^c(f_1 \;\mathsf{U}\; f_2) = [(c \rightarrow \mathcal{T}^c(f_1)) \;\mathsf{U}\; (c \wedge \mathcal{T}^c(f_2))]\,]$
6. $\mathcal{T}^c(\{r\}(f)) = \{\mathcal{T}^c(r)\}([\neg c \;\mathsf{U}\; (c \wedge \mathcal{T}^c(f)\,])$
7. $\mathcal{T}^c(\{r_1\} \mapsto \{r_2\}!) = \{\mathcal{T}^c(r_1)\} \mapsto \{\mathcal{T}^c(r_2)\}!)$
8. $\mathcal{T}^c(\{r_1\} \mapsto \{r_2\}) = \{\mathcal{T}^c(r_1)\} \mapsto \{\mathcal{T}^c(r_2)\})$
9. $\mathcal{T}^c(f \text{ abort } b) = \mathcal{T}^c(f) \text{ abort } (c \wedge b)$
10. $\mathcal{T}^c(f@c_1!) = [\neg c_1 \;\mathsf{U}\; (c_1 \wedge \mathcal{T}^{c_1}(f))]$

B.6 Status of the formal semantic definition

The formal semantics presented above contain three anomalies, described below. They will be addressed in version 1.1. A preliminary version of the proposed version 1.1 semantics can be found at: http://www.eda.org/vfv/docs/truncated_semantics.pdf.

1. The strength of the clock has a minimal effect, in that it distinguishes only between paths with no ticks of the clock and paths with one or more ticks of the clock. Thus, for instance, (eventually! b)@clk evaluates to true if there are no ticks of clk, but to false if there's just one tick, at which b doesn't hold. This issue will be addressed by defining the semantics of unclocked Sugar formulas over empty as well as non-empty paths. This will eliminate the need for two clock operators of varying strengths. From the user's point of view, this will have minimal effect, since it is a corner case resulting from a multi-clocked trace "ending too early" after a clock domain switch. To minimize the impact of this change, tool builders can use the rewrite rules to implement the clock operators, and then move to the new rewrite rules in the next version.

2. As pointed out by Armoni et al. in http://www.cs.rice.edu/ vardi/misc/ abortreset.pdf, the complexity of the abort operator is problematical. This issue will be addressed by modifying the semantics of abort to what Armoni et al. term "reset semantics". Thus, in the semantics presented above the formula $((\mathit{eventually!}\ \mathit{false})\ \mathit{abort}\ b)$ must fail in all designs, while in version 1.1, this formula will pass if b is asserted. From the user's point of view, this will have minimal effect, since it is a corner case resulting from aborting a non-satisfiable formula. From a tool builder's point of view, this will involve removing one step in the algorithm that builds the automaton for a given formula (the step that removes states from which there is no accepting run).

3. An issue related to #2 above involves what happens to weak suffix implications in which the right-hand side contains a SERE whose language is empty. According to the semantics presented above, such a formula will fail in all designs. While in the case of SEREs there is no complexity issue, the semantics for version 1.1 may be modified in such a way that such a formula can pass in some circumstances. This will align the treatment of formulas containing an unsatisfiable sub-formula with the treatment of SEREs whose language is empty. For instance, according to the semantics presented above, the formula $\{a; b; c\} \mapsto \{d; e[*]; \mathit{false}\}$ fails in all designs, since there is no word in the language of $\{d; e[*]; \mathit{false}\}$. In version 1.1, the semantics may be modified to allow this formula to pass in the case that the trace ends with an e, or if there are

an infinite number of e's. From the user's point of view, this will have minimal effect since it is a corner case resulting from coding a SERE whose language is empty. 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 the a given SERE (the step that removes states from which there is no accepting run).

Appendix C

(informative)

Bibliography

[B1] The IEEE Standard Dictionary of Electrical and Electronics Terms, Sixth Edition.

[B2] *EDL Informal Description*, IBM, July 4, 2001.

Index

I

K

L

M

N

O

P

R