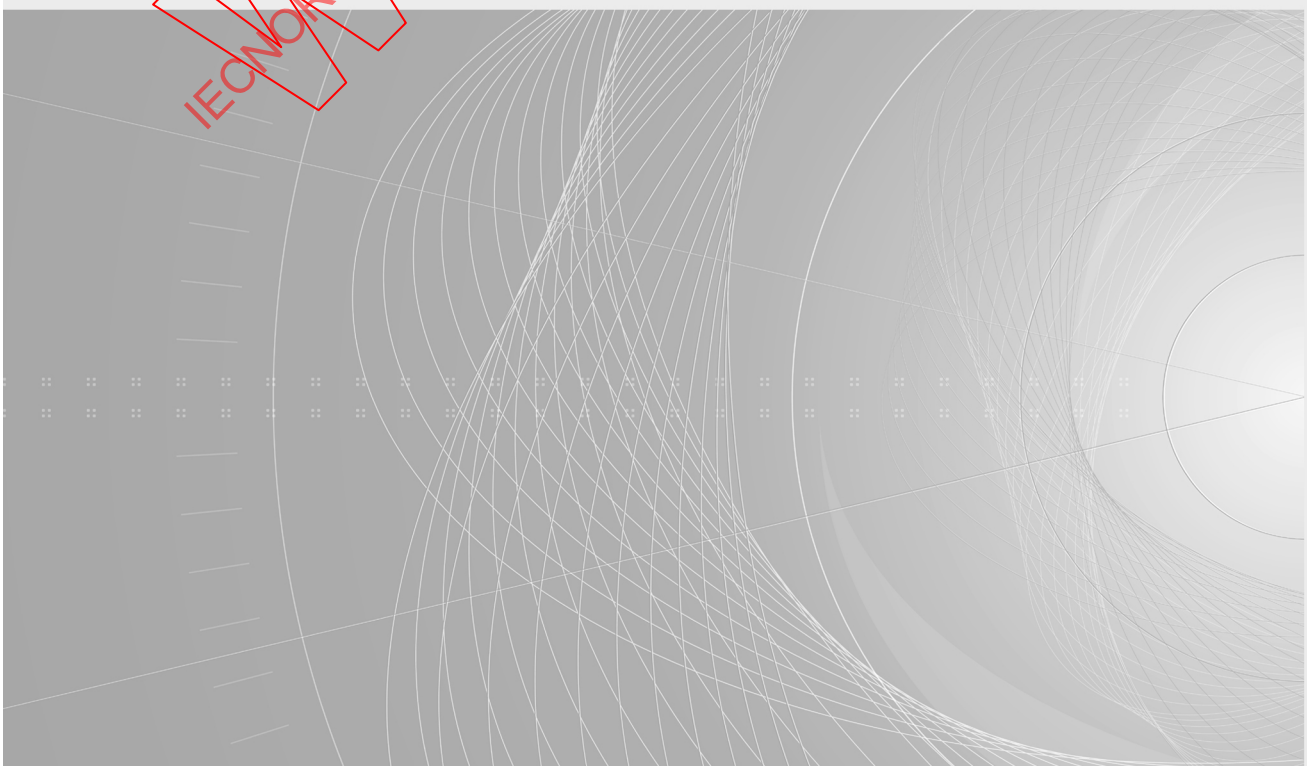
IEC 62531

Edition 1.0 2007-11

INTERNATIONAL STANDARD

IEEE 1850™

Standard for Property Specification Language (PSL)



IEC 62531:2007(E) IEEE Std. 1850-2005

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland
Email: inmail@iec.ch
Web: www.iec.ch

The Institute of Electrical and Electronics Engineers, Inc
3 Park Avenue
US-New York, NY10016-5997
USA
Email: stds-info@ieee.org
Web: www.ieee.org

IEC 62531

Edition 1.0 2007-11

IEEE 1850™

INTERNATIONAL STANDARD

Standard for Property Specification Language (PSL)

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

PRICE CODE XG

ICS 25.040

ISBN 2-8318-9352-6

CONTENTS

INTERNATIONAL ELECTROTECHNICAL COMMISSION

STANDARD FOR PROPERTY SPECIFICATION LANGUAGE (PSL)

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation. IEC collaborates closely with the International Organization for Standardization (ISO) in accordance with conditions determined by agreement between the two organizations.
2) The formal decisions or agreements of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees.
3) IEC Publications have the form of recommendations for international use and are accepted by IEC National Committees in that sense. While all reasonable efforts are made to ensure that the technical content of IEC Publications is accurate, IEC cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.
4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC Publication and the corresponding national or regional publication shall be clearly indicated in the latter.
5) IEC provides no marking procedure to indicate its approval and cannot be rendered responsible for any equipment declared to be in conformity with an IEC Publication.
6) Attention is drawn to the possibility that some of the elements of this IEC Publication may be the subject of patent rights. IEC shall not be held responsible for identifying any or all such patent rights.

International Standard IEC/IEEE 62531 has been processed through Technical Committee 93: Design automation.

The text of this standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1850(2005)	93/253/FDIS	93/264/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

The committee has decided that the contents of this publication will remain unchanged until the maintenance result date indicated on the IEC web site under "http://webstore.iec.ch" in the data related to the specific publication. At this date, the publication will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IEC/IEEE Dual Logo International Standards

This Dual Logo International Standard is the result of an agreement between the IEC and the Institute of Electrical and Electronics Engineers, Inc. (IEEE). The original IEEE Standard was submitted to the IEC for consideration under the agreement, and the resulting IEC/IEEE Dual Logo International Standard has been published in accordance with the ISO/IEC Directives.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an IEC/IEEE Dual Logo International Standard is wholly voluntary. The IEC and IEEE disclaim liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEC or IEEE Standard document.

The IEC and IEEE do not warrant or represent the accuracy or content of the material contained herein, and expressly disclaim any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEC/IEEE Dual Logo International Standards documents are supplied "AS IS".

The existence of an IEC/IEEE Dual Logo International Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEC/IEEE Dual Logo International Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEC and IEEE are not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Neither the IEC nor IEEE is undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEC/IEEE Dual Logo International Standards or IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations – Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of IEC/IEEE Dual Logo International Standards are welcome from any interested party, regardless of membership affiliation with the IEC or IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA and/or General Secretary, IEC, 3, rue de Varembé, PO Box 131, 1211 Geneva 20, Switzerland.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

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

IEEE Standard for

Property Specification Language (PSL)

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

and the
IEEE Standards Association Corporate Advisory Group

Approved 22 September 2005

IEEE-SA Standards Board

Grateful acknowledgment is made to Accellera Organization, Inc. for the permission to use the following source material:

Accellera Property Specification Language Reference Manual (version 1.1), Accellera

GDL: General Description Language, Accellera, Mar. 2005

Abstract: The IEEE Property Specification Language (PSL) is defined in this standard. PSL is a formal notation for specification of electronic system behavior, compatible with multiple electronic system design languages, including IEEE Std 1076™ (VHDL®), IEEE Std 1364™ (Verilog®), IEEE P1666™ (SystemC®), and IEEE P1800™ (SystemVerilog®), thereby enabling a common specification and verification flow for multi-language and mixed-language designs. PSL captures design intent in a form suitable for simulation, formal verification, formal analysis, and hybrid verification tools. PSL enhances communication among architects, designers, and verification engineers to increase productivity throughout the design and verification process. The primary audiences for this standard are the implementors of tools supporting the language and advanced users of the language.
Keywords: ABV, assertion, assertion-based verification, assumption, cover, model checking, property, PSL, specification, temporal logic, verification

IEEE Introduction

IEEE Std 1850 Property Specification Language (PSL) is based upon the Accellera Property Specification Language (Accellera PSL), a language for formal specification of electronic system behavior, which was developed by Accellera, a consortium of Electronic Design Automation (EDA), semiconductor, and system companies. IEEE Std 1850 PSL refines Accellera PSL version 1.1, addressing errata and a few minor technical issues and clarifying how PSL interfaces with various standard electronic system design languages.

Notice to users

Errata

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

Interpretations

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

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention. A patent holder or patent applicant has filed a statement of assurance that it will grant licenses under these rights without compensation or under reasonable rates and nondiscriminatory, reasonable terms and conditions to applicants desiring to obtain such licenses. The IEEE makes no representation as to the reasonableness of rates, terms, and conditions of the license agreements offered by patent holders or patent applicants. Further information may be obtained from the IEEE Standards Department.

STANDARD FOR
PROPERTY SPECIFICATION VOLTAGE (PSL)

1. Overview

1.1 Scope

This standard defines the property specification language (PSL), which formally describes electronic system behavior. This standard specifies the syntax and semantics for PSL and also clarifies how PSL interfaces with various standard electronic system design languages.

1.2 Purpose

The purpose of this standard is to provide a well-defined language for formal specification of electronic system behavior, one that is compatible with multiple electronic system design languages, including IEEE Std 1076™ (VHDL), IEEE Std 1364™ (Verilog®), IEEE P1800™[1] (SystemVerilog®), and IEEE P1666™ (SystemC), to facilitate a common specification and verification flow for multi-language and mixed-language designs.

1.2.1 Background

The complexity of Very Large Scale Integration (VLSI) has grown to such a degree that traditional approaches have begun to reach their limitations, and verification costs have reached 60%–70% of development resources. The need for advanced verification methodology, with improved observability of design behavior and improved controllability of the verification process, has become critical. Over the last decade, a methodology based on the notion of “properties” has been identified as a powerful verification paradigm that can assure enhanced productivity, higher design quality and, ultimately, faster time to market and higher value to engineers and end-users of electronics products. Properties, as used in this context, are concise, declarative, expressive and unambiguous specifications of desired system behavior, that are used to guide the verification process. IEEE Std 1850 PSL is a standard language for specifying electronic system behavior using properties. PSL facilitates property-based verification using both simulation and formal verification, thereby enabling a productivity boost in functional verification.

[1]Numbers preceded by P are IEEE authorized standards projects that were not approved by the IEEE-SA Standards Board at the time this publication went to press. For information about obtaining drafts, contact the Institute of Electrical and Electronics Engineers, Inc.

1.2.2 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 IEEE 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. In addition, it provides a standard means for hardware designers and verification engineers to create a rigorous and machine-executable design specification.

1.2.3 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 specifications of a large class of real-world design properties
— Known efficient underlying algorithms in simulation, as well as formal verification

1.3 Usage

PSL is a language for the formal specification of hardware. It is used to describe properties that are required to hold in the design under verification. PSL provides a means to write specifications that are both easy to read and mathematically precise. It is intended to be used for functional specification on the one hand and as input to functional verification tools on the other. Thus, a PSL specification is an executable specification 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 three kinds of elements: *Boolean expressions*, which describe behavior over one cycle; *sequential expressions*, which can describe multi-cycle behavior; and *temporal operators*, which describe

temporal relationships among Boolean expressions and sequences. For example, consider the following Verilog Boolean expression:

```
ena || enb
```

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

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

describes a sequence of cycles, such that `req` is asserted in the first cycle, `ack` is asserted in the second cycle, and `cancel` is deasserted in the third cycle. The following property, obtained by applying the temporal operators `always` and `|=>` to these expressions,

```
always {req;ack;!cancel} |=> (ena || enb)
```

means that always (that is, in every cycle), if the sequence `{req;ack;!cancel}` occurs, then either `ena` or `enb` is asserted one cycle after the sequence ends. Adding the directive `assert` as follows:

```
assert always {req;ack;!cancel} |=> (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 in the subclauses that follow.

1.3.2.1 Simulation

A PSL specification can also be used to automatically generate checks of simulated behavior. 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 during simulation.

For instance, the following PSL property:

```
Property 1: always (req -> next !req)
```

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

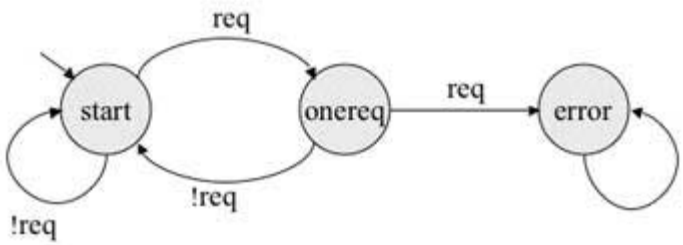


Figure 1—A simple (deterministic) FSM that checks Property 1

For properties more complicated than the property shown in Figure 1, manually writing a corresponding checker is painstaking and error-prone, and maintaining a collection of such checkers for a constantly changing design under development is a time-consuming task. Instead, a PSL specification can be used as input to a tool that automatically generates simulatable checkers.

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

Property 2: `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.

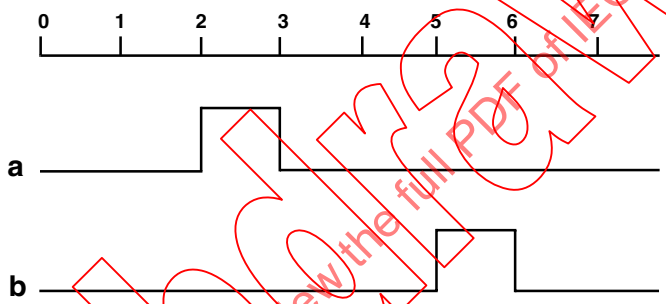


Figure 2—A trace that satisfies Property 2

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

Property 3: `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.

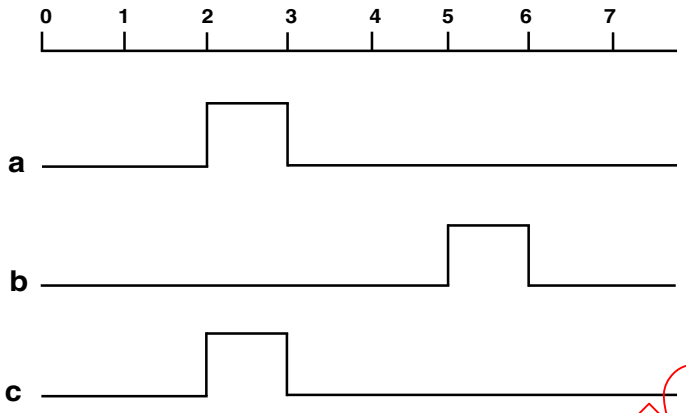


Figure 3—A trace that satisfies Property 3

1.3.2.2 Formal verification

PSL is an extension of the standard temporal logics Linear-Time Temporal Logic (LTL) and Computation Tree Logic (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*.

2. Normative references

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

"General Description Language," Accellera, Napa, CA, Mar. 2005.[2]

IEC/IEEE 62142 (IEEE Std 1364.1), Standard for Verilog Register Transfer Level Synthesis.[3]

IEEE Std 1076™, IEEE Standard VHDL Language Reference Manual.[4] [5]

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

IEEE Std 1364™, IEEE Standard for Verilog Hardware Description Language.

IEEE P1666™, Draft Standard for the SystemC Language.

IEEE P1800™, Draft Standard for the SystemVerilog Language.

[2]This document is available from the IEEE Standards World Wide Web site, at http://standards.ieee.org/downloads/1850/1850-2005/gdl.pdf.

[3]IEC publications are available from the Sales Department of the International Electrotechnical Commission, Case Postale 131, 3, rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iec.ch/). IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

[4]IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (http://standards.ieee.org/).

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

3. Definitions, acronyms, and abbreviations

For the purposes of this standardt, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B5][6] should be referenced for terms not defined in this clause.

3.1 Definitions

This subclause 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 functional 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 functional verification tools to consider only paths on which the given property holds.

3.1.3 asynchronous property: A property whose clock context is equivalent to True.

3.1.4 behavior: A path.

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 term used to identify the last cycle of a path that satisfies a sequential expression or property.

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

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

3.1.12 cycle: An evaluation cycle.

3.1.13 describes: A term used to identify the set of behaviors for which 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.

[6]The numbers in brackets correspond to those of the bibliography in Annex C.

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

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

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

3.1.19 extension (of a given path): A path that starts with precisely the succession of states in the given path.

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

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

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

3.1.23 functional 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.

3.1.24 holds: A term used to talk about the meaning of a Boolean expression, sequential expression, or property.

3.1.25 holds tightly: A term used to talk about the meaning of a sequential expression. Sequential expressions are evaluated over finite paths (behavior).

3.1.26 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.

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

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

3.1.29 model checking: A type of formal verification.

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

3.1.31 occurs: A term used to indicate that a Boolean expression holds in a given cycle.

3.1.32 occurrence (of a Boolean expression): a 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 (of a given path): 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 behaviors.

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. A range may describe a set of values or a variable number of cycles or event repetitions.

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

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

3.1.42 sequence: A sequential expression that may be used directly within a property or directive.

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

3.1.44 sequential extended regular expression: A form of sequential expression, and a component of a sequence.

3.1.45 starts: A term used to identify the first cycle of a path that satisfies a sequential expression.

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

3.1.47 strong operator: A temporal operator, the non-negated use of which usually creates a liveness property.

3.1.48 temporal expression: An expression that involves one or more temporal operators.

3.1.49 temporal operator: An operator that represents a temporal (i.e., time-oriented) relationship between its operands.

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 SystemVerilog and Verilog flavors, the single bit value `1'b1` is interpreted as the logical value *True*. In the VHDL flavor, the values `STD.Standard.Boolean'(True)`, `STD.Standard.Bit'('1')`, `IEEE.std_logic_1164.std_logic'('1')`, and `IEEE.std_logic_1164.std_logic'('H')` interpreted as the logical value *True*. In the SystemC flavor, the value `'true'` of type bool and any integer literal with a non-zero numeric value are interpreted as the logical value *True*. In the GDL flavor, the Boolean value `'true'` and bit value `1B` are both interpreted as the logical value *True*.

3.1.53 unknown value: A value of a (multi-valued) logic type, other than 0 or 1.

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 subclause lists the acronyms and abbreviations used in this standard.

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

3.3 Special terms

The following terms are used in the definition of this standard.

When presenting requirements, options, and recommendations regarding the implementation and use of PSL, the following terms are used:

— **can:** Used for statements of possibility and capability. In the context of this standard, describes a possible use of PSL to express a given specification, or a possible application of a PSL specification in the design and verification of electronic systems.
— **may:** Used to indicate a course of action permissible within the limits of the standard. In the context of this standard, typically describes a non-mandatory feature of PSL syntax or semantics, the use of which in a given PSL specification is up to the author of that specification. Also used to identify permissible implementation approaches in a verification tool supporting the standard, as well as permissible decisions that can be made when implementing a design according to a given PSL specification.
— **shall:** Used to indicate mandatory requirements to be followed strictly in order to conform to the standard and from which no deviation is permitted. In the context of this standard, describes a mandatory feature of PSL syntax or semantics that must be present in a given PSL specification, or in the negative form, a syntactic structure or semantic relationship that must not be present, for that specification to be in conformance with the standard.
— **should:** Used to indicate that among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited. In the context of this standard, describes a feature of PSL syntax that is recommended but not mandatory, or (in the negative form) that is not recommended but not prohibited.

When explaining the requirements and options imposed by a PSL specification on the behavior of a design or a design's environment, if that design or environment is to satisfy the PSL specification, the following term is used:

— **is required to:** Used to indicate that the functionality or behavior described by a PSL specification is mandatory for the system to which the specification pertains. This phrase is typically used to state that a design or its environment must function in a manner that is consistent with the specification.

4. Organization

4.1 Abstract structure

PSL consists of four layers, which partition the language with respect to functionality. PSL also comes in five flavors, which partition the language with respect to HDL compatibility. Each of these is explained in detail in the following subclauses.

4.1.1 Layers

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

4.1.1.1 Boolean layer

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

4.1.1.2 Temporal layer

The temporal 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

The verification 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

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

4.1.2.1 SystemVerilog flavor

In the SystemVerilog flavor, all expressions of the Boolean layer, as well as modeling layer code, are written in SystemVerilog syntax (see IEEE P1800[7]). The SystemVerilog flavor also has limited influence on the

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

syntax of the temporal layer. For example, ranges of the temporal layer are specified using the SystemVerilog-style syntax `i:j`.

4.1.2.2 Verilog flavor

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

4.1.2.3 VHDL flavor

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

4.1.2.4 SystemC flavor

In the SystemC flavor, all expressions of the Boolean layer, as well as modeling layer code, are written in SystemC syntax (see IEEE P1666). The SystemC flavor also has limited influence on the syntax of the temporal layer. For example, ranges of the temporal layer are specified using the syntax `i:j`.

4.1.2.5 GDL flavor

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

4.2 Lexical structure

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

4.2.1 Identifiers

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

Example

```
mutex
Read_Transaction
L_123
```

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

4.2.2 Keywords

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

The keywords used in PSL are shown in Table 1.

Table 1—Keywords

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

[a] **and** is a keyword only in the VHDL flavor; see the flavor macro `AND_OP` (4.3.2.6).
[b] **is** is a keyword only in the VHDL flavor; see the flavor macro `DEF_SYM` (4.3.2.9).
[c] **not** is a keyword only in the VHDL flavor; see the flavor macro `NOT_OP` (4.3.2.6).
[d] **or** is a keyword only in the VHDL flavor; see the flavor macro `OR_OP` (4.3.2.6).
[e] **to** is a keyword only in the VHDL flavor; see the flavor macro `RANGE_SYM` (4.3.2.7).

4.2.3 Operators

4.2.3.1 HDL operators

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

4.2.3.2 Foundation Language (FL) operators

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

Table 2—FL operator precedence and associativity

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

NOTE—The notation `next*` represents the *next* family of operators, which includes the operators `next`, `next!`, `next_a`, `next_a!`, `next_e`, `next_e!`, `next_event`, `next_event!`, `next_event_a!`, and `next_event_e!`. The notation `until*` represents the *until* family of operators, which includes the operators `until`, `until!`, `until_`, and `until!_`. The notation `before*` represents the *before* family of operators, which includes the operators `before`, `before!`, `before_`, and `before!_`.[8]

4.2.3.2.1 Union operator

For any flavor of PSL, the FL operator with the next highest precedence after the HDL operators is that used to indicate a non-deterministic expression:

`union` union operator

The union operator is left-associative.

4.2.3.2.2 Clocking operator

For any flavor of PSL, the FL operator with the next highest precedence is the clocking operator, which is used to associate a clock expression with a property or sequence:

`@` clock event

The clocking operator is left-associative.

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

4.2.3.2.3 SERE repetition operators

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

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

SERE repetition operators are left-associative.

4.2.3.2.4 Sequence within operator

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

`within`	sequence within operator

The sequence within operator is left-associative.

4.2.3.2.5 Sequence conjunction operators

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

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

Sequence conjunction operators are left-associative.

4.2.3.2.6 Sequence disjunction operator

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

`|`	sequence disjunction

The sequence disjunction operator is left-associative.

4.2.3.2.7 Sequence fusion operator

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

`:`	sequence fusion

The sequence fusion operator is left-associative.

4.2.3.2.8 Sequence concatenation operator

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

`;`	sequence concatenation

The sequence concatenation operator is left-associative.

4.2.3.2.9 FL termination operators

For any flavor of PSL, the FL operators with the next highest precedence are the FL termination operators, which are used to describe behavior in which a condition causes both current and future obligations to be canceled:

`sync_abort`	immediate termination of current and future obligations, synchronous with the clock
`async_abort`	immediate termination of current and future obligations, independent of the clock
`abort`	equivalent to async_abort

The FL termination operators are left-associative.

4.2.3.2.10 FL occurrence operators

For any flavor of PSL, the FL operators with the next highest precedence are those used to describe behavior in which an operand holds in the future. These operators are:

`eventually!`	the right operand holds at some time in the indefinite future
`next*`	the right operand holds at some specified future time or range of future times

FL occurrence operators are right-associative.

4.2.3.2.11 Bounding operators

For any flavor of PSL, the FL operators with the next highest precedence are those used to describe behavior in which one property holds in some cycle or in all cycles before another property holds. These operators are:

`until*`	the left operand holds at every time until the right operand holds
`before*`	the left operand holds at some time before the right operand holds

FL bounding operators are right-associative.

4.2.3.2.12 Suffix implication operators

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

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

The suffix implication operators are right-associative.

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

4.2.3.2.13 Logical implication operators

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

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

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

4.2.3.2.14 FL invariance operators

For any flavor of PSL, the FL operators with the next highest precedence are those used to describe behavior in which a property does or does not hold, globally. These operators are:

`always`	the right operand holds, globally
`never`	the right operand does NOT hold, globally

FL occurrence operators are right-associative.

4.2.3.3 Optional Branching Extension (OBE) operators

Table 3—OBE operator precedence and associativity

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

4.2.3.3.1 OBE occurrence operators

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

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

The OBE occurrence operators are left-associative.

4.2.3.3.2 OBE implication operators

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

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

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

4.2.4 Macros

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

4.2.4.1 The %for construct

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

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

or:

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

The replicator name `var` is any legal PSL identifier name. It shall not be the same as any other identifier (variable, unit name, design signal etc.) except another non-enclosing PSL replicator var. The replication expressions `expr1` and `expr2` shall be statically computed expressions resulting in a legal PSL range. A replication item `item` is any legal PSL alphanumeric string or previously defined cpp style macro.

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

a) 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;
```

b) If `ii` is part of an identifier, the value of `ii` may be accessed using `%{ii}` as follows:

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

which is equivalent to:

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

c) If `ii` needs to be used as part of an expression, it may 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 may be used in pre-processor expressions:

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

4.2.4.2 The %if construct

The `%if` construct is similar to the `#if` construct of the cpp pre-processor. However, unlike the `#if` construct, the `%if` construct can be conditioned on variables defined in an enclosing `%for` construct. 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 SystemC, SystemVerilog, and Verilog flavors, both the block comment style (`/* .... */`) and the trailing comment style (`//` .... <*eol*>) are supported.

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

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

4.3 Syntax

4.3.1 Conventions

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

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

PSL_Statement

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

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

vunit (;

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

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

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

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

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

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

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

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

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

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

g) A colon (:) in a production starts a line comment unless it appears in boldface, in which case it stands for itself.

h) If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *vunit*_Name is equivalent to Name.

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

4.3.2 HDL dependencies

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

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

Example

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

shows the range symbol macro (RANGE_SYM).

PSL also defines a few extensions to Verilog declarations as shown in Syntax 4-1.

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

Syntax 4-1—Extended Verilog Declaration

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 Syntax 4-2.

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

Syntax 4-2—Flavor macro HDL_UNIT

4.3.2.2 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 Syntax 4-3.

```
Flavor Macro HDL_DECL  =
     SystemVerilog: SystemVerilog_module_or_generate_item_declaration
   / Verilog: Extended_Verilog_Declaration
   / VHDL: VHDL_declaration
   / SystemC: SystemC_declaration
   / GDL: GDL_module_item_declaration

Flavor Macro HDL_STMT  =
     SystemVerilog: SystemVerilog_module_or_generate_item
   / Verilog: Verilog_module_or_generate_item
   / VHDL: VHDL_concurrent_statement
   / SystemC: SystemC_statement
   / GDL: GDL_module_item
```

Syntax 4-3—Flavor macros HDL_DECL and HDL_STMT

4.3.2.3 HDL_EXPR and HDL_CLOCK_EXPR

Expressions in PSL are those allowed in the underlying HDL description. This applies to expressions appearing directly within a temporal layer property, including those that appear within clock expressions, as well as to any sub-expressions of those expressions. The definitions of HDL_EXPR and HDL_CLOCK_EXPR capture this requirement, as shown in Syntax 4-4.

```
Flavor Macro HDL_EXPR =
      SystemVerilog: SystemVerilog_Expression
    / Verilog: Verilog_Expression
    / VHDL: VHDL_Expression
    / SystemC: C++_Expression
    / GDL: GDL_Expression

Flavor Macro HDL_CLOCK_EXPR =
      SystemVerilog: SystemVerilog_Event_Expression
    / Verilog: Verilog_Event_Expression
    / VHDL: VHDL_Expression
    / SystemC: SystemC_Event_Expression
    / GDL: GDL_Expression

SystemC_Event_Expression ::=
      sc_event
    | sc_event_finder
    | sc_event_and_list
    | sc_event_or_list
    | sc_signal
    | sc_port
```

Syntax 4-4—Flavor macro HDL_EXPR and HDL_CLOCK_EXPR

4.3.2.4 HDL_VARIABLE_TYPE

The formal types of PSL named declarations may be HDL variable types. PSL formal types are described in 6.3.1. Flavor macro HDL_VARIABLE_TYPE defines the HDL types that may be used as PSL formal type, as shown in Syntax 4-5.

```
Flavor Macro HDL_VARIABLE_TYPE =
      SystemVerilog : SystemVerilog_data_type
    / Verilog : Verilog_Variable_Type
    / VHDL : VHDL_subtype_indication
    / SystemC: SystemC_simple_type_specifier
    / GDL : GDL_variable_type

Verilog_Variable_Type ::=
      task_port_type
    | reg [ signed ] [ range ]
```

Syntax 4-5—Flavor macro HDL_VARIABLE_TYPE

4.3.2.5 HDL_RANGE

Some HDLs provide special syntax for referring to the range of values that a variable or index may take on. Flavor macro HDL_RANGE captures this possibility, as shown in Syntax 4-6. Unlike other flavor macros, this one only includes options for those languages that support special range syntax.

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

Syntax 4-6—Flavor macro HDL_RANGE

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

4.3.2.6 AND_OP, OR_OP, and NOT_OP

Each flavor of PSL overloads the underlying HDL's symbols for the logical 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 Syntax 4-7.

```
Flavor Macro AND_OP =
    SystemVerilog: && / Verilog: && / VHDL: and / SystemC: && / GDL: &

Flavor Macro OR_OP =
    SystemVerilog: || / Verilog: || / VHDL: or / SystemC: || / GDL: |

Flavor Macro NOT_OP =
    SystemVerilog: ! / Verilog: ! / VHDL: not / SystemC: ! / GDL: !
```

Syntax 4-7—Flavor macros AND_OP, OR_OP, and NOT_OP

4.3.2.7 RANGE_SYM, MIN_VAL, and MAX_VAL

Within properties it is possible to specify a range of integer values representing the number of cycles, or number of repetitions that are allowed to occur, or a range of integer values to specify the set of values in a for or forall property. 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 Syntax 4-8.

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

Flavor Macro MIN_VAL =
    SystemVerilog: 0 / Verilog: 0 / VHDL: 0 / SystemC: 0 / GDL: null

Flavor Macro MAX_VAL =
    SystemVerilog: $ / Verilog: inf / VHDL: inf / SystemC: inf / GDL: null
```

Syntax 4-8— Flavor macros RANGE_SYM, MIN_VAL, and MAX_VAL

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

4.3.2.8 LEFT_SYM and RIGHT_SYM

In replicated properties, it is possible to specify the replication index `Name` as a vector of Boolean values. PSL allows this specification to take the form of an array reference in the underlying HDL, as reflected in the definition of `LEFT_SYM` and `RIGHT_SYM` shown in Syntax 4-9.

```
Flavor Macro LEFT_SYM =
    SystemVerilog: [ / Verilog: [ / VHDL: ( / SystemC: [ / GDL: (

Flavor Macro RIGHT_SYM =
    SystemVerilog: ] / Verilog: ] / VHDL: ) / SystemC: ] / GDL: )
```

Syntax 4-9—Flavor macro LEFT_SYM and RIGHT_SYM

4.3.2.9 DEF_SYM

Finally, as in the underlying HDL, PSL can declare new named objects. To make the syntax of such declarations consistent with those in the HDL, PSL adopts the symbol used for declarations in the underlying HDL, as reflected in the definition of `DEF_SYM` shown in Syntax 4-10.

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

Syntax 4-10—Flavor macro DEF_SYM

4.4 Semantics

The following subclauses introduce various general concepts related to temporal property specification and explain how they apply to PSL.

4.4.1 Clocked vs. unclocked evaluation

Every PSL property, sequence, and built-in function has an associated clock context. The property, sequence, or built-in function is evaluated only in cycles in which the clock context holds. A nested property, sequence, or built-in function within a given property or sequence can have a different clock context than that of the parent property or sequence.

The *base clock context* is True, i.e., the granularity of time as seen by the verification tool. Different verification tools may model time at different levels of granularity. For example, an event-driven simulation tool typically has a relatively fine-grained model of time, whereas a cycle-based simulation or formal verification tool typically has a more coarse-grained model of time.

A clock context may be specified locally, or may be inherited from an enclosing construct, or may be specified by a default clock declaration. A locally specified clock context takes precedence over an inherited or default clock context.

A PSL property or sequence whose clock context is specified locally by a clock expression associated with the property or sequence (by the @ operator) is a *clocked* property or sequence, respectively; otherwise it is an *unclocked* property or sequence.

A PSL property, sequence, or built-in function whose clock context is equivalent to True is an *asynchronous* property, sequence, or built-in function; otherwise it is a *synchronous* property, sequence, or built-in function, respectively.

A synchronous PSL property that contains no nested asynchronous properties, sequences, or built-in functions shall give the same result in cycle-based and event-based verification tools, provided that there is a one-to-one, in-order correspondence between (a) the succession of event-based states in which any clock context of the property holds, and (b) the succession of cycle-based states in which the same clock context holds.

4.4.2 Safety vs. liveness properties

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

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

4.4.3 Linear vs. branching logic

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

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

4.4.4 Simple subset

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

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

— The operand of a negation operator is a Boolean.
— The operand of a `never` operator is a Boolean or a Sequence.
— The operand of an `eventually!` operator is a Boolean or a Sequence.
— At most one operand of a logical or operator is a non-Boolean.
— The left-hand side operand of a logical implication (`->`) operator is a Boolean.

— Both operands of a logical iff (`<->`) operator are Boolean.
— The right-hand side operand of a non-overlapping `until*` operator is a Boolean.
— Both operands of an overlapping `until*` operator are Boolean.
— Both operands of a `before*` operator are Boolean.
— The operand of `next_e*` is Boolean.
— The FL Property operand of `next_event_e*` is Boolean.

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

4.4.5 Finite-length vs. infinite-length behavior

The semantics of PSL allow us to decide whether a PSL property holds on a given behavior. How the outcome of this problem relates to the design depends on the behavior that was analyzed. In dynamic verification, only behaviors that are finite in length are considered. In such a case, PSL defines the following four levels of satisfaction of a property:

Holds strongly:

— No bad states have been seen
— All future obligations have been met
— The property will hold on any extension of the path

Holds (but does not hold strongly):

— No bad states have been seen
— All future obligations have been met
— The property may or may not hold on any given extension of the path

Pending:

— No bad states have been seen
— Future obligations have not been met
— (The property may or may not hold on any given extension of the path)

Fails:

— A bad state has been seen
— (Future obligations may or may not have been met)
— The property will not hold on any extension of the path

4.4.6 The concept of strength

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

4.4.6.1 Strong vs. weak operators

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

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

The strong forms of such operators are sometimes referred to as *unbounded strong operators*. The unbounded strong operators of PSL include operators of the following families: `F`, `eventually!`, `U`, `until!`, `until!_`, `before!`, `before!_`, and `next_event!`.

Some operators have a requirement about the minimal length of the examined path. For example, the property "signal ack must follow signal req within 3 cycles" is expressed using an operator of the `next` family, which states that there should be at least 3 clock cycles after signal req is asserted, and that signal ack should be asserted at one of these cycles.

Such operators also come in both strong and weak forms. The strong form requires that the examined path be long enough for the condition to occur, while the weak form requires the condition to occur, only if the examined path is indeed long enough (if the path is not long enough, there is no requirement for the condition to hold). For example, the strong and weak forms of *signal ack must follow signal req within 3 cycles* are (`req -> next_e![1 to 3] ack`) and (`req -> next_e[1 to 3] ack`), respectively. The former states that signal ack should be asserted at one of the 3 cycles after signal req is asserted, and in particular that the path should consist between 1 to 3 clocks cycles after signal req is asserted (the exact number depends on when ack holds). The latter states that if there are at least 3 clock cycles after signal req is asserted, then signal ack should be asserted at one of those 3 cycles.

The strong forms of such operators are sometimes referred to as *bounded strong operators*. The bounded strong operators of PSL include operators of the following families: `X!`, `next!`, `next_a!`, and `next_e!`.

Not all strong operators of PSL can be classified as bounded or unbounded. The syntactic form `Sequence!` is a strong syntactic form and it is bounded if Sequence may not take unbounded time, and is unbounded otherwise.

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

4.4.6.2 Strong satisfaction

Strong satisfaction is related to the status of a property on a finite path, as seen, for example, in simulation. If a property holds on a finite path, and in addition, the property is one that would hold on any extension of that path, then it is said that the property holds strongly (see 4.4.5), or equivalently, that it is satisfied strongly. For instance, the property (expressed in English) *`p` is eventually asserted* holds strongly on a finite path on which `p` is asserted at some point. The property (expressed in English) *`p` is always asserted* does not hold strongly on such a path (and indeed holds strongly on no finite path), because extending the path could cause the property to fail.

4.4.6.3 Relating the two concepts of strength

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

5. Boolean layer

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

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

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

HDL-specific expressions are not redefined by PSL. Rather, PSL uses a subset of the existing IEEE standards. The details of this subset are given in 5.1.

5.1 Expression type classes

Five classes of expression are distinguished in PSL: Bit, Boolean, BitVector, Numeric, and String expressions. Each of these corresponds to a set of specific data types in the underlying HDL context, and an interpretation of the values of those data types.

Some PSL built-in functions take operands that may be of any HDL data type or PSL type class, as shown in Syntax 5-11. In such a case, there is no interpretation of type or values involved.

```
Any_Type ::=
  HDL_or_PSL_Expression
```

Syntax 5-11—Any type expression

Other PSL built-in functions and expressions, and PSL temporal layer constructs, require operands that belong to specific type classes. In such a case, if an HDL expression appears in a location at which the PSL grammar requires an expression of a specific PSL type class, then the value of the HDL expression will be interpreted as a value of a corresponding PSL type class, as described below.

PSL expressions and built-in functions can be used in an HDL context, either in the modeling layer or in an HDL expression within the modeling layer. In such a case, the value of the PSL type class returned by the PSL expression or built-in function is converted back to a specific HDL data type, as described below.

If an HDL expression appears immediately within an HDL context, e.g., as a subexpression within another HDL expression, then neither the interpretation of HDL expression values as values of a PSL type class, nor the conversion of values of a PSL type class back to values of an HDL data type, apply.

5.1.1 Bit expressions

Bit expressions represent the values of individual signals or memory elements in the design. The data types used in bit expressions include types that model bits as strictly binary (having values in {0,1}) as well as multi-valued logic types, with values in {X, 0, 1, Z}. (See Syntax 5-12.)

```
Bit ::=
  bit_HDL_or_PSL_Expression
```

Syntax 5-12—Bit expression

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

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

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

In SystemC, types sc_bit and sc_bv are Bit types.

In GDL, type *boolean* is a Bit type.

5.1.2 Boolean expressions

Boolean expressions, for which the Boolean layer is named, describe states of the design, in terms of signals, values, and their relationships. They represent simple properties, which can be composed using temporal operators to create temporal properties. (See Syntax 5-13.)

```
Boolean ::=
  boolean_HDL_or_PSL_Expression
```

Syntax 5-13—Boolean expression

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

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

In SystemC, type bool is a Boolean type.

In GDL, type *boolean* is a Boolean type.

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

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

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

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

5.1.3 BitVector expressions

BitVector expressions represent words composed of bits, of various widths, as shown in Syntax 5-14.

```
BitVector ::=
  bitvector_HDL_or_PSL_Expression
```

Syntax 5-14—BitVector expression

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

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

In SystemC, each of the types sc_bv, sc_lv, sc_int, sc_uint, sc_bigint, and sc_biguint is interpretable as a BitVector type.

5.1.4 Numeric expressions

Numeric expressions represent integer constants such as cycle or occurrence counts that are part of the definition of a temporal property, as shown in Syntax 5-15.

```
Number ::=
  numeric_HDL_or_PSL_Expression
```

Syntax 5-15—Numeric expression

In Verilog, any BitVector expression that contains no unknown bit values is interpretable as a Numeric expression. In SystemVerilog, any integral type is interpretable as a Numeric type. In VHDL, any expression of an integer type is interpretable as a Numeric expression. In SystemC, any expression of type bool, char, short, int, long, or long long, or of types sc_bit, sc_bv, sc_int, sc_uint, sc_bigint, or sc_biguint, is interpretable as a Numeric expression. In GDL, any expression of an integer type, or of type Boolean, is interpretable as a Numeric expression.

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

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

Restrictions

Numeric expressions shall be statically evaluable—signals or variables that change value over time shall not be used in Numeric expressions. Numeric expressions are always required to be non-negative; in some cases they are required to be non-zero as well.

5.1.5 String expressions

String expressions represent text messages that are attached to a PSL directive to help in debugging, as shown in Syntax 5-16.

```
String ::=
  string_HDL_or_PSL_Expression
```

Syntax 5-16—String expression

In Verilog and GDL, any string literal is a String expression. In SystemVerilog, any expression of type string is a String expression. In VHDL, any expression of type *STD.Standard.String* is a String expression. In SystemC, any expression of type std::string or char* is a String expression.

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

5.2 Expression forms

Expressions in the Boolean Layer are built from HDL expressions, PSL expressions, PSL built-in functions, and union expressions, as Syntax 5-17 illustrates.

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

Syntax 5-17—HDL or PSL Expression

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

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

5.2.1 HDL expressions

An HDL expression may be used wherever a Bit, Boolean, BitVector, Numeric, or String expression is required, provided that the type of the expression is (or is interpretable as) the required type. The form of HDL expression allowed in a given context is determined by the flavor of PSL being used, as shown in Syntax 5-18.)

```
HDL_Expression ::=
  HDL_EXPR

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

Syntax 5-18—HDL expression

Informal Semantics

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

For each name and operator symbol in the HDL expression, the meaning of the name or operator symbol is determined as follows:

a) If this is an operator symbol that is predefined in the flavor of PSL used in this verification unit, then the operator symbol has its predefined meaning.

b) If the current verification unit contains a (single) declaration of this name or operator symbol, then the object created by that declaration is the meaning of this name or operator symbol.

c) 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 or operator symbol, then the object created by that declaration is the meaning of this name or operator symbol.

d) Otherwise, if the default verification mode contains a (single) declaration of this name or operator symbol, then the object created by that declaration is the meaning of this name or operator symbol.

e) Otherwise, if this name or operator symbol has an unambiguous meaning at the end of the design module or instance to which the current verification unit is bound, then that meaning is the meaning of this name or operator symbol.

f) Otherwise, this name or operator symbol has no meaning.

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

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

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

See 7.2 for an explanation of verification units and modes.

5.2.2 PSL expressions

PSL defines a collection of operators that represent underlying HDL operators, as shown in Syntax 5-19.

```
HDL_or_PSL_Expression ::=
   PSL_Expression

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

Syntax 5-19—PSL expression

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

Informal Semantics

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

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

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

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

5.2.3 Built-in functions

PSL defines a collection of built-in functions that detect typically interesting conditions, or compute useful values, as shown in Syntax 5-20.

```
Built_In_Function_Call ::=
     prev ( Any_Type [ , Number [ , Clock_Expression ]] )
   | next ( Any_Type )
   | stable ( Any_Type [ , Clock_Expression ] )
   | rose ( Bit [ , Clock_Expression ] )
   | fell ( Bit [ , Clock_Expression ] )
   | ended ( Sequence [ , Clock_Expression ])
   | isunknown ( BitVector )
   | countones ( BitVector )
   | onehot ( BitVector )
   | onehot0 ( BitVector )
   | nondet ( Value_List )
   | nondet_vector ( Number, Value_List )
```

Syntax 5-20—Built-in functions

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

5.2.3.1 prev()

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

If there is no (i^{th}) previous clock cycle or that clock cycle is not at initialization time or later and if a value is given to the expression for time points prior to initialization time by the simulation semantics for the HDL underlying the PSL flavor in question then the built-in function `prev()` should return that value.

NOTE 1—In the absence of an explicit clock context parameter, the clock context is determined by the context in which the built-in function appears, as defined by the rules given in 5.3 for determination of the clock context of a Boolean (specifically, a built-in function).

NOTE 2—The first argument of `prev()` is not necessarily a Boolean expression. For example, if the argument to `prev()` is a bit vector, then the result is the previous value of the entire bit vector.

Restrictions

If a call to `prev()` includes a Number, it shall 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 its clock context is True. In the context of clock `clk`, the call `prev(a)` returns the value 1 at times 5 and 7, and the value 0 at other tick points. In the context of clock `clk`, the call `prev(a,2)` returns the value 1 at time 7, and 0 at other tick points.

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

5.2.3.2 next()

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

Restrictions

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

Example

In the timing diagram below, the function call `next(a)` returns the value 1 at times 1, 2, and 4, and the value 0 at other times.

```
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
```

The value of `next(a)` is not affected by the clock context (implied here by the signal `clk` in the timing diagram).

Function `next()` can be used to create a signal in the modeling layer that mirrors (i.e., always has the same value as) another signal. This is particularly useful in conjunction with the nondeterministic assignments involving the union operator or the `nondet()` or `nondet_vector()` built-in functions. For example, consider the following code:

```
always @(posedge clk)
    rega <= #1 exp1 union exp2;
```

This assigns a value to rega that is either the value of exp1 or the value of exp2, nondeterministically chosen when the assignment is executed.

Suppose regb is required to have the same value as rega under certain conditions. Assigning the value of `rega` to `regb` would introduce a delay, which might not be acceptable. Assigning the same expression (`exp1 union exp2`) to `regb` would not work, because the assignment to `regb` would also be nondeterministic, and therefore `rega` and `regb` could end up with different values. However, using the `next()` function, the following code would ensure that, whenever the enable input is high, `regb` is always assigned the same value as `rega` is being assigned:

```
always @(posedge clk)
    if (enable) regb <= #1 next(rega);
```

5.2.3.3 stable()

The built-in function `stable()` takes an expression of any type as argument. With a single argument, `stable()` returns True if the argument's value is the same as it was at the previous cycle, with respect to the clock of its context; otherwise, it returns False. If a second argument is specified and has the value `c`, the built-in function `stable()` returns True if the first argument's value is the same as it was at the previous cycle, with respect to clock context `c`; otherwise, it returns False.

The function `stable()` can be expressed in terms of the built-in function `prev()` as follows: For any bit expression `e` and any Boolean `c`, `stable(e,c)` is equivalent to the Verilog or SystemVerilog expression `(prev(e,1,c) === e)`, and is equivalent to the VHDL expression `(prev(e,1,c) = e)`. The function `stable()` may be used anywhere a Boolean is required.

NOTE—If the clock context is True, the clock context is determined by the context in which the built-in function appears, as defined by the rules given in 5.3 for determination of the clock context of a Boolean (specifically, a built-in function).

Example

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

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

5.2.3.4 rose()

The built-in function `rose()` takes a Bit expression as argument. With a single argument, `rose()` returns 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 returns False. If a second argument is specified and has the value `c`, the built-in function `rose()` returns True if the first argument's value is `1` at the current cycle and `0` at the previous cycle, with respect to clock context `c`; otherwise, it returns False.

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

NOTE 1—In the absence of an explicit clock context parameter, the clock context is determined by the context in which the built-in function appears, as defined by the rules for determination of the clock context of a Boolean (specifically, a built-in function), given in 5.3.

NOTE 2—The function rose(c) is similar to the Verilog event expression (posedge c) and the VHDL function rising_edge(c) defined in package IEEE.std_logic_1164. For a given property f and signal clk, `f@rose(clk)`, `f@(posedge clk)`, and `f@(rising_edge(clk))` all have equivalent semantics, provided that signal clk takes on only 0 and 1 values, and no signal in f changes at the same time as clk (i.e., there are no race conditions).

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

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

Example

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

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

5.2.3.5 fell()

The built-in function `fell()` takes a Bit expression as argument. With a single argument, `fell()` returns True if the argument's value is `0` at the current cycle and `1` at the previous cycle, with respect to the clock of its context; otherwise, it returns False. If a second argument is specified and has the value `c`, the built-in function `fell()` returns True if the first argument's value is `1` at the current cycle and `0` at the previous cycle, with respect to clock context `c`; otherwise, it returns False.

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

NOTE 1—In the absence of an explicit clock context parameter, the clock context is determined by the context in which the built-in function appears, as defined by the rules given in 5.3 for determination of the clock context of a Boolean (specifically, a built-in function).

NOTE 2—The function fell(c) is similar to the Verilog event expression (negedge c) and the VHDL function falling_edge(c) defined in package IEEE.std_logic_1164. For a given property f and signal clk, `f@fell(clk)`, `f@(negedge clk)`, and `f@(falling_edge (clk))` all have equivalent semantics, provided that signal clk takes on only 0 and 1 values, and no signal in f changes at the same time as clk (i.e., there are no race conditions).

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

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

Example

In the timing diagram below, the function call `fell(a)` is true at times 4 and 6 and at no other time if its clock context is True. In the context of clock `clk`, the function call `fell(a)` is true at the tick of `clk` at time 7 and at no other tick point of `clk`.

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

5.2.3.6 ended()

The built-in function `ended()` takes a Sequence as an argument. With a single argument, `ended()` returns True in any cycle in which the sequence completes; otherwise it returns False. If the first argument is `s`, and a second argument `c` is specified, then it is equivalent to `ended({s}@c)`. Function `ended()` may be used anywhere a Boolean is required.

NOTE—In the absence of an explicit clock context parameter, the clock context is determined by the context in which the built-in function appears, as defined by the rules given in 5.3 for determination of the clock context of a Boolean (specifically, a built-in function).

5.2.3.7 isunknown()

The built-in function `isunknown()` takes a BitVector as argument. It returns True if the argument contains any bits that have unknown values; otherwise it returns False.

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

5.2.3.8 countones()

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

Bits that have unknown values are ignored.

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

5.2.3.9 nondet()

The built-in function `nondet()` takes one Value Set argument. The set of values can be specified in four different ways:

— The keyword **boolean** specifies the set of values {True, False}.
— A Value Range specifies the set of all Number values within the given range.
— A comma (,) between Value Ranges indicates the union of the obtained sets.
— A list of comma-separated values specifies a value set of arbitrary type; all values must be of the same underlying HDL type.

The function `nondet()` performs nondeterministic choice among the values in the Value Set, and returns the chosen value. The value returned is of the same type as the Value Set elements.

If the type of the return value is *T*, then the function `nondet()` may be used anywhere that a value of type *T* is allowed.

Examples

```
nondet(boolean)          -- returns a value chosen nondeterministically in the
                         -- set {True, False}

nondet( {1:2,4,15:18} )  -- returns a value chosen nondeterministically
                         -- in the set {1,2,4,15,16,17,18}
```

5.2.3.10 nondet_vector()

This function accepts two arguments. The first argument is a Number. The second argument is a Value Set, as specified for the `nondet()` function. If the first argument to `nondet_vector()` is *k*, it returns an array of length *k*, whose elements are chosen nondeterministically in the set of values described by the second argument.

If the type of the Value Set elements is *T*, then the function `nondet_vector()` may be used anywhere that an array of length *k* with elements of type *T* is allowed.

The first argument of `nondet_vector()` must be a positive Number that is statically evaluatable.

Examples

```
nondet_vector(16, boolean)-- returns an array of length 16, with each element
                          -- chosen nondeterministically in the set {True, False}

nondet(8, {1:2,4,15:18})  -- returns an array of length 8, with each element chosen
                          -- nondeterministically in the set {1,2,4,15,16,17,18}
```

5.2.3.11 onehot(), onehot0()

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

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

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

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

5.2.4 Union expressions

The union operator specifies two values, shown in Syntax 5-21, either of which can be the value of the resulting expression.

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

Syntax 5-21—Union expression

Restrictions

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

Example

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

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

5.3 Clock expressions

A clock expression determines when other expressions (including temporal expressions) are evaluated (see Syntax 5-22).

```
Clock_Expression :=
      boolean_Name
    | boolean_Built_In_Function_Call
    | ( Boolean )
    | ( HDL_CLOCK_EXPR )

Flavor Macro HDL_CLOCK_EXPR =
      SystemVerilog: SystemVerilog_Event_Expression
    / Verilog: Verilog_Event_Expression
    / VHDL: VHDL_Expression
    / SystemC: SystemC_Expression
    / GDL: GDL_Expression
```

Syntax 5-22—Clock expression

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

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

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

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

In the SystemC flavor, any expression that SystemC allows to be used as the condition in an if statement may be used as a clock expression. In addition, any SystemC event expression may be used as a clock expression. Such a clock expression is considered to hold in a given cycle iff it generates an event in that cycle.

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

Informal Semantics

A clock expression defines a clock context. A clock context determines the path on which an FL Property, Sequence, or Boolean is evaluated.

The path determined by a given clock context consists of the succession of states in which the clock context holds. The base clock context is True, which holds in every cycle and therefore represents the smallest granularity of time as seen by the verification tool. A clock expression itself must be evaluated on the path determined by the base clock context.

A subordinate FL Property, Sequence, or Boolean may have an explicitly specified clock context that is different from that of the immediately enclosing construct.

For a Boolean, including a built-in function call, the clock context is

— (For a built-in function call only) specified by the optional clock parameter, if present; otherwise
— Inherited from the immediately enclosing Boolean expression or built-in function call, if any; otherwise
— True, if it is the right operand of an abort operator; otherwise
— True, if it appears immediately within a clock expression or in modeling layer code; otherwise
— Inherited from the immediately enclosing property or sequence, if any; otherwise
— True

For a property or sequence, the clock context is

— Specified by the @ operator, if present; otherwise
— Inherited from the immediately enclosing property or sequence, if any; otherwise
— Inherited from the property or sequence in which it is instantiated, if any; otherwise
— (For a top-level property or sequence) specified by the applicable default clock declaration, if any; otherwise
— True

NOTE—The fact that a clock expression must be evaluated on the path determined by the base clock context implies that, if a built-in function call appears in a clock expression and includes a parameter to specify the clock context of the built-in function call, then the value of that parameter must be equivalent to True.

5.4 Default clock declaration

A *default clock declaration*, shown in Syntax 5-23, specifies the clock context of the top-level property or sequence of any directive to which the default declaration applies.

```
PSL_Declaration ::=
    Clock_Declaration

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

Syntax 5-23—Default clock declaration

Restrictions

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

Informal Semantics

The applicable default clock declaration is determined as follows:

a) If the current verification unit contains a (single) default clock declaration, then that is the applicable default clock declaration.
b) Otherwise, if the transitive closure with respect to inheritance of all verification units inherited by the current verification unit contains a (single) default clock declaration, then that is the applicable default clock declaration.
c) Otherwise, if the default verification mode contains a (single) default clock declaration, then that is the applicable default clock declaration.

d) Otherwise, no applicable default clock declaration exists.

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

Example

```
default clock = (posedge clk);

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

is equivalent to

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

NOTE 1—A property f@True, in the context of a default clock, has the same effect as property f, without a default clock. The clock expression True effectively masks the default clock so that it has no effect on property f.

NOTE 2—The default clock declaration

```
default clock = True ;
```

has the same effect as having no default clock declaration.

6. Temporal layer

The temporal layer is used to define sequential expressions and properties, both of which describe behavior over time. Both can describe the behavior of the design or the behavior of the external environment.

A sequential expression is built from the following elements:

— Boolean expressions
— Clock expressions
— Subordinate sequential expressions

A property is built from four types of building blocks:

— Boolean expressions
— Clock expressions
— Sequential expressions
— Subordinate properties

Boolean expressions and clock expressions are part of the Boolean layer; they are described in Clause 5. Sequential expressions involve various forms called Sequential Extended Regular Expressions (SEREs), which are described in 6.1.1. Sequences, a distinguished form of SERE, are described in 6.1.2. Properties are described in 6.2.

In the following subclauses, the term cycle refers to states in which the clock context of the corresponding property, sequence, or Boolean holds, and the term path refers to a succession of zero or more such cycles.

Informal Semantics

Sequential expressions are evaluated over finite paths (see 3.1.33), i.e., behaviors of the design. A sequential expression is said to hold tightly on a given finite path (see 3.1.25, 4.4.5) if the finite path satisfies the sequential expression. Each form of sequential expression is presented in a subclause of 6.1; for each form, the corresponding subclause specifies the cases in which a given finite path satisfies that form of sequential expression.

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

A Boolean expression, sequential expression, or property is evaluated over the first cycle of a finite or infinite path. A Boolean expression, sequential expression, or property is said to hold on a given path (see 3.1.24, 4.4.5) if the path satisfies the Boolean expression, sequential expression, or property. Each form of property is presented in a subclause of 6.2; for each form, the corresponding subclauses specifies the cases in which a given path satisfies that form of property.

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 cycle. Note that the path itself may be of length greater than three. 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.

A Boolean expression, sequential expression, or property is said to describe (see 3.1.13) the set of behaviors that satisfy it; that is, the set of behaviors for which the Boolean expression, sequential expression, or property holds. A Boolean expression is said to occur (see 3.1.31) in a cycle if it holds in that cycle. An occurrence of a Boolean expression (see 3.1.32) is a cycle in which that Boolean expression occurs, or holds. For example, "the next occurrence of b" refers to the next cycle in which the Boolean expression `b` holds.

A sequential expression is said to start at the first cycle of any behavior for which it holds. In addition, a sequential expression starts at the first cycle of any behavior that is a 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. A sequential expression is said to complete 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. A Boolean condition that causes a property to complete is called a terminating condition. A property that causes another property to complete is called a terminating property.

6.1 Sequential expressions

6.1.1 Sequential Extended Regular Expressions (SEREs)

SEREs shown in Syntax 6-24, describe single- or multi-cycle behavior built from a series of Boolean expressions.

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

Syntax 6-24—SEREs and Sequences

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

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

A sequential expression is evaluated on a path, which is defined by the clock context of the sequential expression and by the clock contexts of any subordinate sequential expression. See 5.3 for an explanation of how the clock context of a sequential expression, or portion thereof, is determined.

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

6.1.1.1 Simple SEREs

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

6.1.1.1.1 SERE concatenation (;)

The *SERE concatenation* operator (**;**), shown in Syntax 6-25, constructs a SERE that is the concatenation of two other SEREs.

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

Syntax 6-25—SERE concatenation operator

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

Restrictions

None.

Informal Semantics

For SEREs `A` and `B`:

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

6.1.1.1.2 SERE fusion (:)

The *SERE fusion* operator (**:**), shown in Syntax 6-26, constructs a SERE in which two SEREs overlap by one cycle. That is, the second starts at the cycle in which the first completes. (See Syntax 6-26.)

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

Syntax 6-26—SERE fusion operator

The operands of `:` are both SEREs.

Restrictions

None.

Informal Semantics

For SEREs `A` and `B`:

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

6.1.1.2 Compound SEREs

Compound SEREs represent a set of one or more threads of subordinate behaviors, starting from the same cycle, and occurring in parallel. (See Syntax 6-27.)

```
SERE ::=
   Compound_SERE

Compound_SERE ::=
     Repeated_SERE
   | Braced_SERE
   | Clocked_SERE
   | Compound_SERE | Compound_SERE
   | Compound_SERE & Compound_SERE
   | Compound_SERE && Compound_SERE
   | Compound_SERE within Compound_SERE
   | Parameterized_SERE
```

Syntax 6-27—Compound SEREs

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

6.1.1.2.1 SERE or (|)

The *SERE or* operator (|), shown in Syntax 6-28, constructs a Compound SERE in which one of two alternative Compound SEREs hold at the current cycle.

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

Syntax 6-28—SERE or operator

The operands of | are both Compound SEREs.

Restrictions

None.

Informal Semantics

For Compound SEREs `A` and `B`:

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

6.1.1.2.2 SERE non-length-matching and (&)

The *SERE non-length-matching and* operator (`&`), shown in Syntax 6-29, constructs a Compound SERE in which two Compound SEREs both hold at the current cycle, regardless of whether they complete in the same cycle or in different cycles.

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

Syntax 6-29—SERE non-length-matching and operator

The operands of `&` are both Compound SEREs.

Restrictions

None.

Informal Semantics

For Compound SEREs `A` and `B`:

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

6.1.1.2.3 SERE length-matching and (&&)

The *SERE length-matching and* operator (`&&`), shown in Syntax 6-30, constructs a Compound SERE in which two Compound SEREs both hold at the current cycle, and furthermore both complete in the same cycle.

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

Syntax 6-30—SERE length-matching and operator

The operands of `&&` are both Compound_SEREs.

Restrictions

None.

Informal Semantics

For Compound_SEREs `A` and `B`:

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

6.1.1.2.4 SERE within

The SERE within operator (`within`), shown in Syntax 6-31, constructs a Compound SERE in which the second Compound SERE holds at the current cycle, and the first Compound SERE starts at or after the cycle in which the second starts, and completes at or before the cycle in which the second completes.

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

Syntax 6-31—SERE within operator

The operands of `within` are both Compound SEREs.

Restrictions

None.

Informal Semantics

For Compound SEREs A and B:

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

6.1.1.2.5 Parameterized SERE

The parameterizing operators, shown in Syntax 6-32, apply a given base operator to a set of compound SEREs obtained by instantiating a base compound SERE once for each possible value or combination of values of the given parameter(s).

```
Compound_SERE ::=
  Parameterized_SERE

Parameterized_SERE::=
  for Parameters_Definition : And_Or_SERE_OP { SERE }

Parameters_Definition ::=
  Parameter_Definition {, Parameter_Definition }

Parameter_Definition ::=
  PSL_Identifier [ Index_Range ] in Value_Set

And_Or_SERE_Op :: =
  && | & | |
```

Syntax 6-32—Parameterized SERE

NOTE—the term “instantiated” is used figuratively. It does not imply that instantiation actually takes place. Whether or not any instantiation does take place depends on the implementation.

The PSL Identifiers are the names of the parameters. A PSL Identifier with an Index Range is an array. The base operator can be either SERE or (|), SERE length-matching and (&&), or SERE non-length matching and (&). The Compound SERE enclosed in braces is the base compound SERE.

For each PSL Identifier, the Value Set defines the set of values that the corresponding parameter or array elements can take on.

The set of values can be specified in four different ways:

— The keyword **boolean** specifies the set of values {True, False}.
— A Value Range specifies the set of all Number values within the given range.
— A comma (,) between Value Ranges indicates the union of the obtained sets.
— A list of comma-separated values specifies a value set of arbitrary type; all values must be of the same underlying HDL type.

If the value set is specified by a list of values of arbitrary type, each of the values must be statically computable.

For a single parameter,

a) If the parameter is not an array, and the set of values has size K, then the obtained set is of size K. Each element in the set is obtained by instantiating the base compound SERE with one of the possible values in the set of values.
b) If the parameter is an array of size N, and the set of values has size K then the obtained set is of size K^N. Each element in the set is obtained by instantiating the base compound SERE with one of the combination of values that can be taken on by the array.

For multiple parameters, the set of values is that obtained by applying the above rules repeatedly, once for each parameter.

Restrictions

The restrictions of the base operator apply to the resulting Compound SERE as specified in the subclauses of the respective base operator 6.1.1.2.1 SERE or (|), 6.1.1.2.2 SERE non-length-matching and (&), and 6.1.1.2.3 SERE length-matching and (&&).

For each parameter definition the following restrictions apply:

— If the parameter name has an associated Index Range, the Index Range shall be specified as a finite Range, each bound of the Range shall be statically computable, and the left bound of the Range shall be less than or equal to the right bound of the Range.
— If a Value is used to specify a Value Range, the Value shall be statically computable.
— If a Range is used to specify a Value Range, the Range shall be a finite Range, each bound of the Range shall be statically computable, and the left bound of the Range shall be less than or equal to the right bound of the Range.
— The parameter name shall be used in one or more expressions in the Property, or as an actual parameter in the instantiation of a parameterized SERE, so that each of the instances of the SERE corresponds to a unique value of the parameter name.

NOTE—The parameter is considered to be statically computable, and therefore the parameter names can be used in a static expression, such as that required by a repetition count.

Informal Semantics

For Compound SERE A:

— `for i in boolean: | {A(i)}` is equivalent to applying '`|`' to the set containing the two Compound SEREs:

```
A(false) and A(true),
```

i.e., is equivalent to the Compound SERE:

```
{A(false)} | {A(true)}
```

— `for i in {j:k}: && A(i)` is equivalent to applying '`&&`' to the set containing the k-j+1 Compound SEREs:

```
A(j), A(j+1), ..., A(k),
```

i.e., is equivalent to the Compound SERE:

```
{A(j)} && {A(j+1)} && ... && {A(k)}
```

— `for i in {j,k,l}: && A(i)` is equivalent to applying '`&&`' to the set containing the 3 Compound SEREs:

```
A(j), A(k), and A(l),
```

i.e., is equivalent to the Compound SERE:

```
{A(j)} && {A(k)} && {A(l)}
```

— `for i[0:1] in boolean: & A(i)` is equivalent to applying '`&`' to the set containing the 4 Compound SEREs:

```
A({false,false}),   A({false,true}),
A({true,false}), and A({true,true}),
```

i.e., is equivalent to the Compound SERE:

```
{A({false,false})} &    {A({false,true})} &
{A({true,false})}  &    {A({true,true})}
```

— `for i[0:2] in {c,d}: | A(i)` is equivalent to applying '`|`' to the set containing the 8 Compound SEREs:

```
A({c,c,c}), A({c,c,d}), A({c,d,c}),    A({c,d,d}),
A({d,c,c}), A({d,c,d}), A({d,d,c}), and A({d,d,d}),
```

i.e., is equivalent to the Compound SERE:

```
{A({c,c,c})} | {A({c,c,d})} | {A({c,d,c})} | {A({c,d,d})} |
{A({d,c,c})} | {A({d,c,d})} | {A({d,d,c})} | {A({d,d,d})}
```

— `for i in {j:k}, l in {m:n}: & A(i,l)` is equivalent to applying `&` to the set containing the (k-j+1)×(n-m+1) Compound SEREs:

```
A(j,m),   A(j,m+1),   ..., A(j,n),
A(j+1,m), A(j+1,m+1), ..., A(j+1,n),
...,
A(k,m),   A(k,m+1),   ..., A(k,n),
```

i.e., is equivalent to the Compound SERE:

```
{A(j,m)}   & {A(j,m+1)}   & ... & {A(j,n)}   &
{A(j+1,m)} & {A(j+1,m+1)} & ... & {A(j+1,n)} &
...                                           &
{A(k,m)}   & {A(k,m+1)}   & ... & {A(k,n)}
```

6.1.2 Sequences

A sequence is a SERE that may appear at the top level of a declaration, directive, or property. (See Syntax 6-33.)

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

Syntax 6-33—Sequences

Sequence Instances are described in 6.3.3.1. The remaining forms of Sequence are described in the following subclauses.

6.1.2.1 SERE consecutive repetition ([*])

The *SERE consecutive repetition* operator ([*]), shown in Syntax 6-34, constructs repeated consecutive concatenation of a given Boolean or Sequence.

```
Repeated_SERE::=
    Boolean [* [ Count ] ]
  | Sequence [* [ Count ] ]
  | [* [ Count ] ]
  | Boolean [+]
  | Sequence [+]
  | [+]

Count ::=
    Number
  | Range

Range ::=
    Low_Bound RANGE_SYM High_Bound

Low_Bound ::=
    Number
  | MIN_VAL

High_Bound ::=
    Number
  | MAX_VAL
```

Syntax 6-34—SERE consecutive repetition operator

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

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

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

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

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

Restrictions

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

Informal Semantics

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

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

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

6.1.2.2 SERE non-consecutive repetition ([=])

The *SERE non-consecutive repetition* operator ([=]), shown in Syntax 6-35, constructs repeated (possibly non-consecutive) concatenation of a Boolean expression.

```
Repeated_SERE ::=
    Boolean [= Count ]

Count ::=
    Number
  | Range

Range ::=
    Low_Bound RANGE_SYM High_Bound

Low_Bound ::=
    Number | MIN_VAL

High_Bound ::=
    Number | MAX_VAL
```

Syntax 6-35—SERE non-consecutive repetition operator

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

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

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

Restrictions

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

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

Informal Semantics

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

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

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

6.1.2.3 SERE goto repetition ([->])

The *SERE goto repetition* operator (**[->]**), shown in Syntax 6-36, constructs repeated (possibly non-consecutive) concatenation of a Boolean expression, such that the Boolean expression holds on the last cycle of the path.

```
Repeated_SERE ::=
   Boolean [->[ positive_Count ] ]

Count ::=
   Number
  | Range

Range ::=
   Low_Bound RANGE_SYM High_Bound

Low_Bound ::=
   Number | MIN_VAL

High_Bound ::=
   Number | MAX_VAL
```

Syntax 6-36—SERE goto repetition operator

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

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

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

Restrictions

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

Informal Semantics

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

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

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

6.1.2.4 Braced SERE

A SERE enclosed in braces is another form of sequence, as shown in Syntax 6-37.[9]

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

Syntax 6-37—Braced SERE

6.1.2.5 Clocked SERE (@)

The *SERE clock* operator (@), shown in Syntax 6-38, provides a way to clock a SERE.

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

Syntax 6-38—SERE clock operator

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

The @ operator specifies that the clock expression that is its right operand defines the clock context of its left operand.

NOTE 1—Default clock declarations (5.4) and the optional clock parameters of certain built-in functions (5.2.3) also specify clock contexts.

Restrictions

None.

Informal Semantics

A sequence {R}@C1 is evaluated on a path P1 determined by clock context C1.

If R contains a subordinate built-in function F with clock context C2, and evaluation of R involves evaluating F in some cycle N of P1, then F is evaluated on a path P2 determined by clock context C2 and ending at N.

If R contains a subordinate sequence {S}@C3, and evaluation of R involves evaluating S at some cycle M of P1, then S is evaluated on path P3 starting at M and determined by clock context C3.

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

NOTE 2—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.

Examples

Example 1

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

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

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

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

Example 2

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

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

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

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

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

6.2 Properties

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

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

6.2.1 FL properties

FL Properties, shown in Syntax 6-39, describe single- or multi-cycle behavior built from Boolean expressions, sequential expressions, and subordinate properties.

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

Syntax 6-39—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.

An FL property is evaluated on a path, which is defined by a clock context. See 5.3 for an explanation of how the clock context of an FL Property is determined.

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

6.2.1.1 Sequential FL properties

Sequential expressions are FL properties, which specify that the behavior described by a sequence occurs.

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

Syntax 6-40—Sequential FL Property

Restrictions

None.

Informal Semantics

For a Sequence S:

- The FL Property S! holds on a given path iff there exists a non-empty prefix of the path on which S holds tightly.
- The FL Property S holds on a given path iff either there exists a prefix of the path on which S holds tightly, or the property S! does not fail on the given path.

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

6.2.1.2 Clocked FL properties

The *FL clock operator* operator (@), shown in Syntax 6-41, provides a way to clock an FL Property.

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

Syntax 6-41—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.

The @ operator specifies that the clock expression that is its right operand defines the clock context of its left operand.

NOTE 1—Default clock declarations (5.4) and the optional clock parameters of certain built-in functions (5.2.3) also specify clock contexts.

Restrictions

None.

Informal Semantics

A property A@C1 is evaluated on a path P1 determined by clock context C1.

If A contains a subordinate built-in function F with clock context C2, and evaluation of A involves evaluating F in some cycle N of P1, then F is evaluated on a path P2 determined by clock context C2 and ending at N.

If A contains a subordinate sequence {S}@C3, and evaluation of A involves evaluating S at some cycle M of P1, then S is evaluated on path P3 starting at M and determined by clock context C3.

If A contains a subordinate property B@C4, and evaluation of A involves evaluating B at some cycle M of P1, then B is evaluated on a path P4 starting at M and determined by clock context C4.

NOTE 2—When clocks are nested, the inner clock takes precedence over the outer clock. That is, the property `(a -> b@clk2)@clk` is equivalent to the property `(a@clk -> b@clk2)`, with the outer clock applied to only the unclocked sub-properties (if any). In particular, there is no conjunction of nested clocks involved.

Example 1

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

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

The unclocked FL Property

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

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

The clocked FL Property

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

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

Example 2

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

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

The unclocked FL Property

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

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

The singly-clocked FL Property

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

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

The singly-clocked FL Property

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

does not hold at any time of the given behavior.

The multiply-clocked FL Property

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

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

6.2.1.3 Simple FL properties

6.2.1.3.1 always

The `always` operator, shown in Syntax 6-42, specifies that an FL Property holds at all times, starting from the present.

```
FL_Property ::=
  always FL_Property
```

Syntax 6-42—always operator

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

Restrictions

None.

Informal Semantics

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

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

6.2.1.3.2 never

The `never` operator, shown in Syntax 6-43, specifies that an FL Property or a sequence never holds.

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

Syntax 6-43—never operator

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

Restrictions

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

Informal Semantics

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

6.2.1.3.3 eventually!

The `eventually!` operator, shown in Syntax 6-44, specifies that an FL Property holds at the current cycle or at some future cycle.

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

Syntax 6-44—eventually! operator

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

Restrictions

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

Informal Semantics

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

6.2.1.3.4 next

The `next` family of operators, shown in Syntax 6-45, 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)
```

Syntax 6-45—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 specify 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 1—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:
  a) There is a next cycle and
  b) 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:
  a) There is not a next cycle or
  b) 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:
  a) There is an i^{th} next cycle and
  b) 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:
  a) There is not an i^{th} next cycle or
  b) The FL Property that is the operand holds at the i^{th} next cycle.

NOTE 2—The property `next(f)` is equivalent to the property `next[1](f)`.

6.2.1.4 Extended next FL properties

6.2.1.4.1 next_a

The `next_a` family of operators, shown in Syntax 6-46, 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 )
```

Syntax 6-46—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:
  a) There is a j^{th} next cycle and
  b) The FL Property that is the operand holds at all cycles between the i^{th} and j^{th} next cycle, inclusive.
— A `next_a[i:j]` property holds in the current cycle of a given path iff the FL Property that is the operand holds at all cycles between the i^{th} and j^{th} next cycle, inclusive. (If not all those cycles exist, then the FL Property that is the operand holds on as many as do exist.)

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

6.2.1.4.2 next_e

The `next_e` family of operators, shown in Syntax 6-47, 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 )
```

Syntax 6-47—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.

Within the simple subset (see 4.4.4), the operand of `next_e` or `next_e!` is restricted to be a Boolean.

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
  a) There are less than j next cycles following the current cycle, or
  b) There is some cycle between the i^{th} and j^{th} next cycle, inclusive, where the FL Property that is the operand holds.

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

6.2.1.4.3 next_event

The `next_event` family of operators, shown in Syntax 6-48, 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 )
```

Syntax 6-48—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:
  a) The Boolean expression and the FL Property that are the operands both hold at the current cycle, or at some future cycle, and
  b) 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:

a) The Boolean expression that is the operand does not hold at the current cycle, nor does it hold at any future cycle; or
b) 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:
a) The Boolean expression that is the operand holds at least *i* times, starting at the current cycle, and
b) 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:
a) The Boolean expression that is the operand does not hold at least *i* times, starting at the current cycle, or
b) The Boolean expression that is the operand holds at least *i* times, starting at the current cycle, and the FL Property that is the operand holds at the i^{th} occurrence of the Boolean expression.

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

6.2.1.4.4 next_event_a

The `next_event_a` family of operators, shown in Syntax 6-49, 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 )
```

Syntax 6-49—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:
  a) The Boolean expression that is the operand holds at least *j* times, starting at the current cycle, and
  b) The FL Property that is the operand holds at the i^{th} through j^{th} occurrences, inclusive, of the Boolean expression.
— A `next_event_a[i:j]` property holds in a given cycle of a given path iff the FL Property that is the operand holds at the i^{th} through j^{th} occurrences, inclusive, of the Boolean expression, starting at the current cycle. If there are less than *j* occurrences of the Boolean expression, then the FL Property that is the operand holds on all of them, starting from the i^{th} occurrence.

6.2.1.4.5 next_event_e

The `next_event_e` family of operators, shown in Syntax 6-50, 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 )
```

Syntax 6-50—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.

Within the simple subset (see 4.4.4), the FL Property of `next_event_e` or `next_event_e!` is restricted to be a Boolean.

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:

a) There are less than *j* next occurrences of the Boolean expression, or
b) There is some cycle during the i^{th} through j^{th} next occurrences of the Boolean expression at which the FL Property that is the operand holds.

6.2.1.5 Compound FL properties

6.2.1.5.1 abort, async_abort, and sync_abort

The `abort`, `async_abort`, and `sync_abort` operators, shown in Syntax 6-51, specify a condition that removes any obligation for a given FL Property to hold. The `sync_abort` operator expects the abort condition to occur in a cycle in which the context clock holds. The `abort` and `async_abort` operators accept asynchronous abort conditions as well.

```
FL_Property ::=
    FL_Property sync_abort Boolean
  | FL_Property async_abort Boolean
  | FL_Property abort Boolean
```

Syntax 6-51—sync_abort, async_abort, and abort operators

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

Restrictions

None.

Informal Semantics

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

— The FL Property that is the left operand holds, or
— The FL Property that is the left operand does not fail (see 4.4.5) prior to the first cycle (of the path defined by the base clock context) in which the Boolean condition that is the right operand holds.

A `sync_abort` property holds in the current cycle of a given path iff:

— The FL Property that is the left operand holds, or
— The FL Property that is the left operand does not fail (see 4.4.5) prior to the first cycle (of the path defined by the clock context of the abort property) in which the Boolean condition that is the right operand holds.

NOTE 1—The `abort` operator is identical to the `async_abort` operator. It is currently maintained in the language for reasons of backward compatibility.

NOTE 2—For asynchronous properties, aborting with `sync_abort` or `async_abort` (or `abort`) is the same.

Example

Using `async_abort` to model an asynchronous interrupt: "A request is always followed by an acknowledge, unless a cancellation occurs. The request and acknowledge signals are sampled at clock `clk`. The cancellation signal may come asynchronously (not in a cycle of `clk`)."

```
always ((req -> eventually! ack) async_abort cancel)@clk;
```

or

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

when the default clock is `clk`.

Using `sync_abort` to model a synchronous interrupt: "A request is always followed by an acknowledge, unless a cancellation occurs. The request, acknowledge, and cancellation signals are sampled at clock `clk`. A rise of the cancellation signal when `clk` does not hold is ignored."

```
always ((req -> eventually! ack) sync_abort cancel)@clk;
```

or

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

when the default clock is `clk`.

6.2.1.5.2 before

The `before` family of operators, shown in Syntax 6-52, 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
```

Syntax 6-52—before operators

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

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

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

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

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

Restrictions

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

Informal Semantics

— A `before!` property holds in the current cycle of a given path iff:
  a) The FL Property that is the left operand holds at the current cycle or at some future cycle, and
  b) 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:
  a) The FL Property that is the left operand holds at the current cycle or at some future cycle, and
  b) 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:
  a) 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
  b) 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:
  a) 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
  b) The FL Property that is the left operand holds before or at the same cycle as the FL Property that is the right operand.

6.2.1.5.3 until

The `until` family of operators, shown in Syntax 6-53, 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
```

Syntax 6-53—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 4.4.4), the right operand of an `until!` or `until` property is restricted to be a Boolean expression, and both the left and right operands of an `until!_` or `until_` property are restricted to be a Boolean expression.

Informal Semantics

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

6.2.1.6 Sequence-based FL properties

6.2.1.6.1 Suffix implication

The *suffix implication* family of operators, shown in Syntax 6-54, specify that an FL Property or sequence holds if some pre-requisite sequence holds.

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

Syntax 6-54—Suffix implication operators

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

Restrictions

None.

Informal Semantics

— A Sequence |-> FL_Property holds in a given cycle of a given path iff:
  a) The Sequence that is the left operand does not hold at the given cycle, or
  b) The FL Property that is the right operand holds in any cycle C such that the Sequence that is the left operand holds tightly from the given cycle to C.
— A Sequence |=> FL_Property holds in a given cycle of a given path iff:
  a) The Sequence that is the left operand does not hold at the given cycle, or
  b) The FL Property that is the right operand holds in the cycle immediately after any cycle C such that the Sequence that is the left operand holds tightly from the given cycle to C.

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

6.2.1.7 Logical FL properties

6.2.1.7.1 Parameterized property

The parameterizing operators, shown in Syntax 6-55, apply a given base operator to a set of FL Properties obtained by instantiating a base FL Property once for each possible value or combination of values of the given parameter(s).

```
FL_Property ::=
  Parameterized_Property

Parameterized_Property ::=
  for Parameters_Definition : And_Or_Property_OP ( FL_Property )

Parameters_Definition ::=
  Parameter_Definition { Parameter_Definition }

Parameter_Definition ::=
  PSL_Identifier [ Index_Range ] in Value_Set

And_Or_Property_OP ::=
  AND_OP | OR_OP
```

Syntax 6-55—Parameterized property

NOTE 1—The term “instantiated” is used figuratively. It does not imply that instantiation actually takes place. Whether or not any instantiation does take place depends on the implementation.

The PSL Identifiers are the names of the parameters. A PSL Identifier with an Index Range is an array. The base operator can be either a logical and or a logical or. The FL Property enclosed in parenthesis is the base FL Property. For each PSL Identifier, the Value Set defines the set of values that the corresponding parameter or array elements can take on.

The set of values can be specified in four different ways:

— The keyword **boolean** specifies the set of values {True, False}.
— A Value Range specifies the set of all Number values within the given range.
— A comma (,) between Value Ranges indicates the union of the obtained sets.
— A list of comma-separated values specifies a value set of arbitrary type; all values must be of the same underlying HDL type.

If the value set is specified by a list of values of arbitrary type, each of the values must be statically computable.

For a single parameter,

a) If the parameter is not an array, and the set of values has size K, then the obtained set is of size K. Each element in the set is obtained by instantiating the base compound SERE with one of the possible values in the set of values.
b) If the parameter is an array of size N, and the set of values has size K then the obtained set is of size K^N. Each element in the set is obtained by instantiating the base compound SERE with one of the combination of values that can be taken on by the array.

For multiple parameters, the set of values is that obtained by applying the above rules repeatedly, once for each parameter.

Restrictions

The restrictions of the base operator, specified in 6.2.1.7.4 and 6.2.1.7.5 respectively, also apply to parameterized property constructed with the corresponding operator. The simple subset restrictions in 4.4.4 also apply. In particular, since the simple subset restricts the logical or operator to have at most one non-Boolean operand, a parameterized property constructed with the logical or operator belongs to the simple subset iff the base FL Property is Boolean.

For each parameter definition the following restrictions apply:

— If the parameter name has an associated Index Range, the Index Range shall be specified as a finite Range, each bound of the Range shall be statically computable, and the left bound of the Range shall be less than or equal to the right bound of the Range.
— If a Value is used to specify a Value Range, the Value shall be statically computable.
— If a Range is used to specify a Value Range, the Range shall be a finite Range, each bound of the Range shall be statically computable, and the left bound of the Range shall be less than or equal to the right bound of the Range.
— The parameter name shall be used in one or more expressions in the Property, or as an actual parameter in the instantiation of a parameterized SERE, so that each of the instances of the SERE corresponds to a unique value of the parameter name.

NOTE 2—The parameter is considered to be statically computable, and therefore the parameter names may be used in a static expression, such as that required by a repetition count.

NOTE 3—Parameterized properties are a generalization of the forall construct (6.2.3). Any property written with forall can be written equivalently using a parameterized logical and operator. The forall construct is currently maintained in the language for reasons of backward compatibility.

Informal Semantics

For FL_Property F:

— `for i in boolean: || (F(i))` is equivalent to the applying `||` to the set containing the two FL Properties:

`F(false)` and `F(true)`,

i.e., is equivalent to the FL Property:

```
(F(false)) || (F(true))
```

— `for i in {j:k}: && (F(i))` is equivalent to applying `&&` to the set containing the k–j+1 FL Properties:

`F(j), F(j+1), ..., F(k)`,

i.e., is equivalent to the FL Property:

```
(F(j)) && (F(j+1)) && ... && (F(k))
```

— `for i in {j,k,l}: && (F(i))` is equivalent to applying `&&` to the set containing the 3 FL Properties:

`F(j), F(k)`, and `F(l)`,
i.e., is equivalent to the FL Property:

```
(F(j)) && (F(k)) && (F(l))
```

— `for i[0:1] in boolean: && (F(i))` is equivalent to applying `&&` to the set containing the 4 FL Properties:

`F({false,false})`, `F({false,true})`,
`F({true,false})`, and `F({true,true})`,

i.e., is equivalent to the FL Property:

```
(F({false,false})) && (F({false,true})) &&
(F({true,false}))  && (F({true,true}))
```

— `for i[0:2] in {c,d}: || (F(i))` is equivalent to applying `||` to the set containing the 8 FL Properties:

`F({c,c,c})`, `F({c,c,d})`, `F({c,d,c})`, `F({c,d,d})`,
`F({d,c,c})`, `F({d,c,d})`, `F({d,d,c})`, and `F({d,d,d})`,

i.e., is equivalent to the FL Property:

```
(F({c,c,c}))||(F({c,c,d}))||(F({c,d,c}))||(F({c,d,d}))||
(F({d,c,c}))||(F({d,c,d}))||(F({d,d,c}))||(F({d,d,d}))
```

— `for i in {j:k}, l in {m:n}: && (F(i,l))` is equivalent to applying '&&' to the set containing the (k–j+1)×(n–m+1) FL Properties:

```
F(j,m),      F(j,m+1),    ...,   F(j,n),
F(j+1,m),    F(j+1,m+1),  ...,   F(j+1,n),
...,
F(k,m),      F(k,m+1),    ...,   F(k,n)
```

i.e., is equivalent to the FL Property:

```
(F(j,m))     && (F(j,m+1))       && ... && (F(j,n)) &&
(F(j+1,m))   && (F(j+1,m+1))     && ... && (F(j+1,n)) &&
... &&
(F(k,m))     && (F(k,m+1))       && ... && (F(k,n))
```

6.2.1.7.2 Logical implication

The *logical implication* operator (`->`), shown in Syntax 6-56, is used to specify logical implication.

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

Syntax 6-56—Logical implication operator

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

In the SystemC flavor, if the operator '`->`' appears in an expression and its left operand is the name of a pointer to an object that has a member whose name is the right operand, then the '`->`' operator is interpreted as the SystemC member operator, not as the logical implication operator.

Restrictions

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

Informal Semantics

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

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

6.2.1.7.3 Logical iff

The *logical iff* operator (`<->`), shown in Syntax 6-57, is used to specify the iff (if and only if) relation between two properties.

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

Syntax 6-57—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 4.4.4), both operands of a logical iff property are restricted to be a Boolean expression.

Informal Semantics

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

— Both FL Properties that are operands hold at the given cycle, or
— Neither of the FL Properties that are operands holds at the given cycle.

6.2.1.7.4 Logical and

The *logical and* operator, shown in Syntax 6-58, is used to specify logical and.

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

Syntax 6-58—Logical and operator

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

Informal Semantics

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

6.2.1.7.5 Logical or

The *logical or* operator, shown in Syntax 6-59, is used to specify logical or.

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

Syntax 6-59—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 4.4.4), at most one operand of a logical or property may be non-Boolean.

Informal Semantics

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

6.2.1.7.6 Logical not

The *logical not* operator, shown in Syntax 6-60, is used to specify logical negation.

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

Syntax 6-60—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 4.4.4), the operand of a logical not property is restricted to be a Boolean expression.

Informal Semantics

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

6.2.1.8 LTL operators

The *LTL operators*, shown in Syntax 6-61, 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 ]
```

Syntax 6-61—LTL operators

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

Table 4—PSL equivalents

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

Restrictions

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

NOTE—The syntax of the `U` and `W` operators requires brackets, e.g., `[p U q]`. For complete equivalence, the corresponding expressions using PSL operators should be parenthesized. For example, `[p U q]` is equivalent to `(p until! q)`, and `[p W q]` is equivalent to `(p until q)`.

6.2.2 Optional Branching Extension (OBE) properties

Properties of the Optional Branching Extension (*OBE*), shown in Syntax 6-62, 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 nondeterminism 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 )
```

Syntax 6-62—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 Syntax 6-63, specifies that an OBE property holds at all next states of the given state.

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

Syntax 6-63—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 Syntax 6-64, specifies that an OBE property holds at the given state and at all future states.

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

Syntax 6-64—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 Syntax 6-65, 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
```

Syntax 6-65—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 Syntax 6-66, 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 ]
```

Syntax 6-66—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 Syntax 6-67, 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
```

Syntax 6-67—EX operator

Restrictions

None.

Informal Semantics

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

6.2.2.2.2 EG operator

The EG operator, shown in Syntax 6-68, 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
```

Syntax 6-68—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 Syntax 6-69, 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
```

Syntax 6-69—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 Syntax 6-70, 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 ]
```

Syntax 6-70—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 Syntax 6-71, is used to specify logical implication.

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

Syntax 6-71—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 Syntax 6-72, is used to specify the *iff* (if and only if) relation between two properties.

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

Syntax 6-72—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 Syntax 6-73, is used to specify logical and.

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

Syntax 6-73—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 Syntax 6-74, is used to specify logical or.

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

Syntax 6-74—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 Syntax 6-75, is used to specify logical negation.

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

Syntax 6-75—OBE not operator

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

Restrictions

None.

Informal Semantics

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

6.2.3 Replicated properties

Replicated properties are specified using the operator `forall`, as shown in Syntax 6-76. The first operand of the replicated property is a `Replicator` and the second operand is a parameterized property.

```
Property ::=
  Replicator Property

Replicator ::=
  forall Parameter_Definition :

Parameter_Definition ::=
  PSL_Identifier [ Index_Range ] in Value_Set

Index_Range ::=
  LEFT_SYM finite_Range RIGHT_SYM

Flavor Macro LEFT_SYM =
  Verilog: [ / SystemVerilog: [ / VHDL: ( / SystemC: ( / GDL: (

Flavor Macro RIGHT_SYM =
  Verilog: ] / SystemVerilog: ] / VHDL: ) / SystemC: ) / GDL: )

Value_Set ::=
  { Value_Range { , Value_Range } }
  | boolean

Value_Range ::=
  Value
  | finite_Range

Range ::=
  Low_Bound RANGE_SYM High_Bound
```

Syntax 6-76—Replicating properties

NOTE 1—The term *replicated property* is used figuratively. It does not imply that replication actually takes place. Whether or not any part of the property is replicated depends on the implementation.

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

a) If the parameter is not an array, then the property is equivalent to a property obtained by the following steps:
   1) Replicating the parameterized property for each value in the set of values, with that value substituted for the parameter (so that the total number of replications is equal to the size of the set of values)
   2) Logically "anding" all of the replications.
b) If the parameter is an array of size N, then the property is equivalent to a property obtained by the following steps:
   1) Replicating the parameterized property 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)
   2) Logically "anding" all of the replications.

Observe that in both cases the meaning of a replicated property is *equivalent* to the replication process. This does not imply that any replication must actually take place.

The set of values can be specified in four different ways:

— The keyword **boolean** specifies the set of values {*True*, *False*}.
— A `Value Range` specifies the set of all Number values within the given range.
— A comma (**,**) between *Value Ranges* indicates the union of the obtained sets.
— A list of comma-separated values specifies a value set of arbitrary type; all values must be of the same underlying HDL type.

Restrictions

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

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

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

If the value set is specified by a list of values of arbitrary type, each of the values must be statically computable.

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

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

A replicator may appear in the declaration of a named property, provided that instantiations of the named property do not appear inside non-replicated properties.

NOTE 2—The parameter defined by a replicator is considered to be statically computable, and therefore the parameter name can be used in a static expression, such as that required by a repetition count.

NOTE 3—Parameterized properties (6.2.3) are a generalization of the forall construct. Any property written with forall can be written equivalently using a parameterized logical and operator. The forall construct is currently maintained in the language for reasons of backward compatibility.

Informal Semantics

— A `forall i in boolean: f(i)` property is equivalent to:
`f(true) && f(false)`

— A `forall i in {j:k} : f(i)` property is equivalent to:
`f(j) && f(j+1) && f(j+2) && ... && f(k)`

— A `forall i in {j,k} : f(i)` property is equivalent to:
`f(j) && f(k)`

— A `forall i[0:1] in boolean : f(i)` property is equivalent to:
`f({false,false}) && f({false,true}) &&`
`f({true ,false}) && f({true ,true})`

— A `forall i[0:2] in {4,5} : f(i)` property is equivalent to:
`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)

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

Illegal:

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

6.3 Property and sequence declarations

A given temporal expression may be applicable in more than one part of the design. In such a case, it is convenient to be able to define the expression once and refer to the single definition wherever the expression applies. Declaration and instantiation of named declarations provide this capability. (See Syntax 6-77.)

```
PSL_Declaration ::=
   Sequence_declaration | Property_declaration

Sequence_declaration ::=
   sequence PSL_Identifier ( Formal_Parameter_List ) ] DEF_SYM Sequence ;

Property_declaration ::=
   property PSL_Identifier ( Formal_Parameter_List ) ] DEF_SYM Property ;

Formal_Parameter_List ::=
   Formal_Parameter { ; Formal_Parameter }

Formal_Parameter ::=
   Param_Spec PSL_Identifier { , PSL_Identifier }

Param_Spec ::=
   const
   | [ const ] Value_Parameter
   | Temporal_Parameter

Value_Parameter ::=
   HDL_Type
   | PSL_Type_Class

HDL_Type ::=
   hdltype HDL_VARIABLE_TYPE

PSL_Type_Class ::=
   boolean | bit | bitvector | numeric | string

Temporal_Parameter ::=
   sequence | property
```

Syntax 6-77—Property declaration

Informal Semantics

The PSL_Identifier following the keyword `sequence` or `property` in the PSL_Declaration is the name of the declaration. The PSL_Identifiers given in the formal parameter list are the names of the formal parameters of the named declaration.

Restrictions

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

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

6.3.1 Parameters

A named declaration can optionally specify a list of formal parameters that may be referenced in the declaration. An instantiation creates an instance of a named declaration and provides actual expressions for formal parameters.

A formal parameter that is a Value_Parameter can be optionally qualified with `const`. The actual expression that maps to a `const` formal parameter shall be statically computable. If no type is specified for a `const` formal parameter, the parameter shall default to Numeric type.

6.3.1.1 PSL formal parameter type classes

A formal parameter of a PSL sequence or property declaration can be defined to accept any expression that is a member of a general class of expression types. A formal parameter may be defined to be of any of the PSL expression type classes defined in 5.1; a formal parameter may also be defined to be of the class of temporal expressions that includes all Sequences, or the class of temporal expressions that includes all Properties.

Table 5—PSL formal parameter type classes

PSL formal parameter type class	Actual expression type
boolean	Boolean expression (refer to 5.1.2)
bit	Bit expression (refer to 5.1.1)
bitvector	BitVector expression (refer to 5.1.3)
numeric	Numeric expression (refer to 5.1.4)
string	String expression (refer to 5.1.5)
sequence	Sequence (refer to 6.1.2)
property	Property (refer to 6.2.1)

NOTE—A many-to-one mapping from HDL types to a PSL formal parameter type class can result in type ambiguity issues in strongly typed languages like VHDL. For example:

```
sequence s (boolean b0, b1) is {b0 = b1};
```

In the VHDL flavor, both bit and std_ulogic map to Boolean type class, but it is an error to pass expressions of type bit and std_ulogic to formal parameters b0 and b1 respectively in this example if the '=' operator is not defined for operands of type bit and std_ulogic.

6.3.1.2 HDL formal parameter types

In addition to language neutral types, PSL allows formal parameters to be of HDL data types. If an HDL data type can be used in a formal parameter declaration of a subroutine defined in the HDL flavor, it may be used as a formal parameter type in a PSL named declaration. This includes user-defined types. The actual parameter to formal parameter mapping rules are the same as for subroutines in that flavor.

HDL formal parameter types must be qualified with the `hdltype` keyword.

Example

VHDL flavor

```
sequence color_is_red (hdltype COLOR c) is {c = RED};
sequence slope_is_1 (hdltype COORDINATE_RECORD c) is {(c.x / c.y) = 1};
```

SystemVerilog flavor

```
sequence color_is_red (hdltype COLOR c) = {c == RED};
sequence slope_is_1 (hdltype COORDINATE_STRUCT c) = ((c.x / c.y) == 1);
```

6.3.2 Declarations

6.3.2.1 Sequence declaration

A sequence declaration defines a sequence and gives it a name.

Restrictions

Formal parameters of a sequence declaration cannot be of parameter type class Property.

Example

```
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 formal 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.

6.3.2.2 Property declaration

A property declaration defines a property and gives it a name.

Example

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

The two declarations have slightly different interfaces (i.e., different formal parameter orders), but they both declare a property called `ResultAfterN`. Each 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.

6.3.3 Instantiation

An instantiation of a PSL declaration creates an instance of the named declaration and provides an actual parameter for each formal parameter. In the instance created by the instantiation, each actual parameter expression in the actual parameter list of the instantiation replaces all references to the formal parameter in the corresponding position of the formal parameter list of the named declaration.

Restrictions

For each formal parameter of the declaration, the instantiation shall provide a corresponding actual expression. For a `const` formal parameter, the actual expression shall be a statically computable expression. The expression type of the actual parameter shall map to the respective formal parameter type according to the rules specified in 6.3.1. Further, the expression obtained after replacing all formals with the actual expression in the declaration expression shall be a legal expression in the language flavor.

6.3.3.1 Sequence instantiation

A sequence instantiation, shown in Syntax 6-78, creates an instance of a named sequence. An instance of a named sequence is also a Sequence (see 6.1.2).

```
Sequence ::=
   Sequence_Instance

Sequence_Instance ::=
   sequence_Name [ ( sequence_Actual_Parameter_List ) ]

sequence_Actual_Parameter_List ::=
   sequence_Actual_Parameter { , sequence_Actual_Parameter }

sequence_Actual_Parameter ::=
   AnyType | Sequence
```

Syntax 6-78—Sequence instantiation

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.

Example

Given the declarations for the sequences `BusArb` and `ReadCycle` in 6.3.2.1,

```
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 }
```

6.3.3.2 Property instantiation

A property instantiation, shown in Syntax 6-79, creates an instance of a named property. An instance of a named property is also a Property (6.2).

FL_Property ::=
*FL_property*_Name [(Actual_Parameter_List)]

OBE_Property ::=
*OBE_property*_Name [(Actual_Parameter_List)]

Actual_Parameter_List ::=
Actual_Parameter { , Actual_Parameter }

Actual_Parameter ::=
AnyType | Sequence | Property

Syntax 6-79—Property instantiation

Informal Semantics

An instance of a named property that is an FL Property is itself an FL Property.

An instance of a named property that is an OBE Property is itself an OBE Property.

An instance of a named property holds at a given evaluation cycle (for an FL Property) or in a given state (for an OBE Property) iff 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 or state, respectively.

Restrictions

If a named property is an FL Property, and it has a formal parameter that is a Property, then in any instance of that named property, the actual parameter corresponding to that formal parameter shall be an FL Property.

If a named property is an OBE Property, and it has a formal parameter that is a Property, then in any instance of that named property, the actual parameter corresponding to that formal parameter shall be an OBE Property.

Example

Given the first declaration for the property `ResultAfterN` in 6.3.2.2,

```
ResultAfterN (write_req, eventually! ack, 3, cancel)
```

is equivalent to

```
always ((write_req -> next[3] (eventually! ack))
    @ (posedge clk) async_abort  cancel)
```

and

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

is equivalent to

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

7. Verification layer

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

7.1 Verification directives

Verification directives give directions to verification tools.

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

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

Syntax 7-80—Verification Directives

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

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

```
Label ::=
   PSL_Identifier
```

Syntax 7-81—Labels

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

7.1.1 assert

The Assert Directive, shown in Syntax 7-82, instructs the verification tool to verify that a property holds.

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

Syntax 7-82—Assert Directive

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

Example

The directive

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

instructs the verification tool to verify that the property

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

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

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

7.1.2 assume

The Assume Directive, shown in Syntax 7-83, instructs the verification tool to constrain the verification (e.g., the behavior of the input signals) so that a property holds.

Assume_Directive ::=
assume Property **;**

Syntax 7-83—Assume Directive

Restrictions

The Property that is the operand of an Assume Directive must be an FL Property or a replicated FL Property.

Example

The directive

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

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

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

holds in the design.

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

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

7.1.3 assume_guarantee

The Assume Guarantee Directive, shown in Syntax 7-84, instructs the verification tool to perform two different but related tasks: one is to perform a verification constrained with the property in question (as described in 7.1.2) the other one is to verify independently that the property holds.

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

Syntax 7-84—Assume Guarantee Directive

An Assume Guarantee Directive may optionally include a character string containing a message to report when the property fails to hold.

Restrictions

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

Example

The directive

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

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

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

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

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

7.1.4 restrict

The Restrict Directive, shown in Syntax 7-85, is a way to constrain design inputs using sequences.

```
Restrict_Directive ::=
  restrict Sequence ;
```

Syntax 7-85—Restrict Directive

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[*]};
```

instructs the verification tool to constrain the verification (e.g., the behavior of the input signals) so that `rst` is low in the first cycle, `rst` is high for the next three cycles, and `rst` is low forever afterwards.

7.1.5 restrict_guarantee

The Restrict Guarantee Directive, shown in Syntax 7-86, instructs the verification tool to perform two different but related tasks: one is to perform a verification constrained with the sequence in question (as described in 7.1.4) the other one is to verify independently that the sequence holds.

```
Restrict_Guarantee_Directive ::=
  restrict_guarantee Sequence [ report String ] ;
```

Syntax 7-86—Restrict Guarantee Directive

A Restrict Guarantee Directive may optionally include a character string containing a message to report when the sequence fails to hold.

Example

The directive

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

instructs the verification tool to constrain the verification (e.g., the behavior of the input signals) so that `rst` is low in the first cycle, `rst` is high for the next three cycles, and `rst` is low forever afterwards. It also instructs the tool to verify, possibly as a separate step, that the sequence holds. How this information is used is tool-dependent.

NOTE—The optional character string has no effect when a restrict_guarantee directive is being used only to indicate a restriction. The character string can be provided so that it can be reported when the sequence fails to hold in another context.

7.1.6 cover

The Cover Directive, shown in Syntax 7-87, directs the verification tool to check if a certain path was covered by the verification space based on a simulation test suite or a set of given constraints.

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

Syntax 7-87—Cover Directive

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

Example

The directive

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

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

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

7.1.7 fairness and strong_fairness

The Fairness Directives, shown in Syntax 7-88, are special kinds of assumptions that correspond to liveness properties.

```
Fairness_Directive ::=
    fairness Boolean ;
  | strong fairness Boolean , Boolean ;
```

Syntax 7-88—Fairness Directives

If a Fairness Directive 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 a second Boolean expression occurs infinitely many times along the path.

Examples

The directive

```
fairness p;
```

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

```
assume G F p;
```

The directive

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

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

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

7.2 Verification units

A *verification unit*, shown in Syntax 7-89, is used to group verification directives and other PSL statements.

```
Verification_Unit ::=
    Vunit_Type PSL_Identifier [ ( Hierarchical_HDL_Name ) ] {
        { Inherit_Spec }
        { Vunit_Item }
    }

Vunit_Type ::=
    vunit | vprop | vmode

Hierarchical_HDL_Name ::=
    HDL_Module_NAME { Path_Separator instance_Name }

HDL_Module_Name ::=
    HDL_Module_Name [ ( HDL_Module_Name )]

Path_Separator ::=
    . | /

instance_Name ::=
    HDL_or_PSL_Identifier

Inherit_Spec ::=
    inherit vunit_Name { , vunit_Name } ;

Vunit_Item ::=
    HDL_DECL
  | HDL_STMT
  | PSL_Declaration
  | PSL_Directive
```

Syntax 7-89—Verification unit

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

If a Hierarchical HDL Name is given, then the verification unit is explicitly bound to the design module or module instance indicated by the HDL module name(s) and HDL instance name(s) of the Hierarchical HDL Name. If only one HDL module name is given, then the name indicates a VHDL entity, a Verilog module, a SystemVerilog module or interface, or a SystemC (C++) class that inherits from sc_module. If two HDL module names are given, then the pair of module names indicates a VHDL entity and architecture, respectively. If no Hierarchical HDL Name is given, then the verification unit is not explicitly bound. See 7.2.1 for a discussion of binding.

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

A Vunit Item can be any of the following:

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

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

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

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

Restrictions

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

A Verification Unit of type vprop shall not contain a directive that is not an assert directive.

A Verification Unit of type vprop shall not inherit a Verification Unit of type vunit or vmode.

A Verification Unit of type vmode shall not inherit a Verification Unit of type vunit or vprop.

A default Verification Unit, if it exists, shall be of type vmode. The default vmode shall not inherit other verification units of any type.

7.2.1 Verification unit binding

The connection between signals referred to in a verification unit and signals of the design under verification is by name, relative to the module or module instance to which the verification unit is bound.

If the verification unit is explicitly bound to an instance, then HDL names and operator symbols defined in that context may be referenced within the verification unit, as defined in 5.2.1.

If the verification unit is explicitly bound to a module, then this is equivalent to duplicating the contents of the verification unit and binding each duplication to one instance. Observe that the verification unit itself is not duplicated.

If the verification unit is not explicitly bound, then a verification tool may allow the user to specify the binding of the verification unit separate from the verification unit. A verification unit that is not explicitly bound can also be used to group together commonly used PSL declarations so they can be inherited for use in other PSL verification units.

Examples

```
vunit ex1a(top_block.i1.i2) {
  A1: assert never (ena && enb);
}
```

`vunit ex1a` is bound to instance `top_block.i1.i2`. This causes assertion A1 to apply to signals `ena` and `enb` in `top_block.i1.i2`.

As a second example consider

```
vunit ex2a(mod1) {
      assert never (ena && enb);
}
```

The verification unit is bound to `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 vunit:

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

As a third example, consider:

```
vunit ex3 {
  A3: assert never (ena && enb);
}
```

This verification unit is not explicitly bound, so there is no context in which to interpret references to `ena` and `enb`. In this case, the verification tool may determine the binding.

Finally, consider:

```
vunit ex4 {
      property mutex (boolean b1, b2) = never (b1 && b2);
}
```

This verification unit is not explicitly bound; however, it contains no HDL expressions that require interpretation, so no binding is necessary. This illustrates use of an unbound verification unit for commonly used PSL declarations that can be inherited into other verification units.

7.2.2 Verification unit inheritance

One verification unit may inherit one or more other verification units, each of which may inherit other verification units, and so on. Inheritance is transitive.

For a verification unit that inherits one or more other units, its inherited context is the set of verification units in the transitive closure with respect to inheritance. A verification unit must not be contained in its own inherited context.

Inheritance has the following two effects:

a) As a consequence of the rules for determining the meaning of names (in 5.2.1), any PSL declarations in a given verification unit's inherited context can be referenced in that verification unit.
b) When a given verification unit is considered by a verification tool, the contents of that verification unit and its inherited context are taken together to define the environment in which verification is to take place and the set of directives to consider during verification.

Examples

Assume there are two blocks, A and B, which are mutually dependent—the outputs of A (`Aout1`, `Aout2`) are inputs of B (`Bin1`, `Bin2`), and vice versa. The following verification units might describe the interactions between the two blocks:

```
vmode Common {
      property mutex (boolean b1, b2) = never b1 && b2 ;
      property one_hot (boolean b1, b2) =
                  always ((b1 &&! b2) || (b2 && !b1));
}
```

Verification unit Common is not explicitly bound. It contains commonly used property definitions. It is defined as a vmode verification unit so it can be inherited by other vmode units.

```
vmode Amode (blockA) {
      inherit Common;
      assume mutex(Aout1, Aout2);
}
vmode Bmode (blockB) {
      inherit Common;
      assume one_hot(Bout1, Bout2);
}
```

Verification units `Amode` and `Bmode` contain assumptions about these blocks to be made when verifying properties in other blocks. They are both explicitly bound, so that the HDL name references they contain have meaning. They both inherit the Common vmode in order to make use of the property declarations it contains.

```
vunit Aprops (blockA) {
      inherit Common, Bmode;
      assert mutex(Aout1, Aout2);
}
```

```
vunit Bprops (blockB) {
      inherit Common, Amode;
      assert one_hot(Bout1, Bout2);
}
```

Verification units `Aprops` and `Bprops` contain assertions to verify about the respective blocks. They are both explicitly bound, so that the HDL name references they contain have meaning. They both inherit the Common vmode in order to make use of its property declarations. The vunit `Aprops` inherits vmode `Bmode` so that assumptions about the outputs of B can be considered when verifying the behavior of block A. The vunit `Bprops` inherits vmode `Amode` so that assumptions about the outputs of A can be considered when verifying the behavior of block B.

7.2.3 Verification unit scoping rules

A verification unit may contain HDL declarations, including declarations of signal names that are also declared in the design module or instance to which the verification unit is bound. This allows a verification unit to import a design signal written in one HDL into a verification unit written using another HDL flavor. It also allows a verification unit to give new behavior to a signal in the design under verification.

Informal Semantics

If a design signal name is redeclared in a verification unit bound to a given module or instance, and no assignment is made to that name in that verification unit, then the declaration in the verification unit is implicitly assigned the value of the corresponding design signal at all times. If the design is written in one HDL, and the verification unit is written in a different HDL flavor, then standard conventions for mixed-language simulation are used to translate from the type system of the design's HDL to the type system of the verification unit's HDL.

If a design signal name is redeclared in a verification unit bound to a given module or instance, and that name is assigned a value in that verification unit, then the declaration in the verification unit overrides the declaration in the design module or instance and in any inherited verification units bound to the same design module or instance, and the assignment to that name in the verification unit overrides all assignments to the signal with that name in the design and in any inherited verification units bound to the same design module or instance.

Examples

Consider the following verification unit:

```
vunit ex5a(top_block.i1) {
    wire reqa, temp;
 A5a: assert always (reqa -> next temp);
}
```

Verification unit `ex5a` redeclares signals `reqa` and `temp` in a Verilog-flavor verification unit. If the instance to which this verification unit is bound (`top_block.i1`) is a VHDL design, and that instance contains declarations of signals `reqa` and `temp`, then the value of each VHDL signal will be implicitly assigned to the corresponding Verilog signal declared in the vunit, and in the process its value will be translated from the VHDL data type to the corresponding Verilog data type according to the conventions applied in mixed-language simulation.

Now consider the following verification unit:

```
vunit ex5b(top_block.i1) {
      wire temp;
      assign temp = ack1 || ack2;
      A5b: assert always (reqa -> next temp);
}
```

Verification unit `ex5b` declares wire temp and also assigns it a value. This could be just an auxiliary statement to facilitate specification of assertion `A5b`. However, if instance `top_block.i1` also contains a declaration of a signal named '`temp`', then the declaration in `ex5b` would override the declaration in the design, and the assignment to '`temp`' in vunit `ex5b` would override the driving logic for signal '`temp`' in the design.

Now consider the following verification unit:

```
vunit ex5c(top_block.i1) {
      inherit ex5b;
      wire temp;
      assign temp = ack1 || ack2 || ack3;
      A5c: assert always ((reqa || reqb) -> next temp);
}
```

Verification unit `ex5c` inherits `ex5b`. Both verification units are bound to the same instance and both declare wires named temp. The declaration of temp in the inheriting verification unit takes precedence, so the declaration of (and assignment to) `temp` in `ex5c` takes precedence when verifying assertion `A5c`, and the declaration of (and assignment to) `temp` in both the design and vunit `ex5b` are ignored.

Finally, consider the following verification unit:

```
vunit ex5d(top_block.i1) {
      wire reqa;
      assign reqa = nondet({0,1});
      A5c: assert always ((reqa || reqb) -> next temp);
}
```

Verification unit `ex5d` redeclares signal `reqa`, and it also assigns a nondeterministic value of 0 or 1 to the redeclared signal. This causes `reqa` to behave as a free variable, both in the design and in verifying the assertion `A5d`.

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 five flavors, corresponding to SystemVerilog, Verilog, VHDL, SystemC, and GDL.

The SystemVerilog flavor of the modeling layer will consist of the synthesizable subset of SystemVerilog, which is not yet defined.

The Verilog flavor of the modeling layer consists of the synthesizable subset of Verilog, defined by IEEE Std 1364.1. The Verilog flavor of the modeling layer extends Verilog to include integer range declarations, as defined in 8.1, and struct declarations, as defined in 8.2.

The VHDL flavor of the modeling layer consists of the synthesizable subset of VHDL, defined by IEEE Std 1076.6.

The SystemC flavor of the modeling layer consists of those SystemC declarations that would be legal in the context of the SystemC module to which the vunit is bound, and those statements that would be legal in the context of the constructor of the SystemC module to which the vunit is bound.

The GDL flavor of the modeling layer consists of all of GDL.

In each flavor of the modeling layer, at any place where an HDL expression may appear, the modeling layer is extended to allow any form of HDL or PSL expression, as defined in Clause 5. Thus, HDL expressions, PSL expressions, built-in functions, and union expressions may all be used as expressions within the modeling layer.

Each flavor of the modeling layer supports the comment constructs of the corresponding hardware description language.

8.1 Integer ranges

The Verilog flavor of the modeling layer is extended to include declaration of a finite integer type, shown in Syntax 8-90, where the range of values that the variable can take on is indicated by the declaration.

```
Extended_Verilog_Type_Declaration ::=
    Finite_Integer_Type_Declaration

Finite_Integer_Type_Declaration ::=
    integer Integer_Range list_of_variable_identifiers ;

Integer_Range ::=
    ( constant_expression : constant_expression )
```

Syntax 8-90—Integer range declaration

The nonterminals list_of_variable_identifiers and constant_expression are defined in the syntax for IEEE Std 1364.

Example

```
integer (1:5) a, b[1:20];
```

This declares an integer variable `a`, which can take on values between `1` and `5`, inclusive, and an integer array `b`, each of whose twenty entries can take on values between `1` and `5`, inclusive.

8.2 Structures

The Verilog flavor of the modeling layer is also extended to include declaration of C-like structures, as shown in Syntax 8-91.

```
Extended_Verilog_Type_Declaration ::=
      Structure_Type_Declaration

Structure_Type_Declaration ::=
      struct { Declaration_List } list_of_variable_identifiers ;

Declaration_List  ::=
    HDL_Variable_or_Net_Declaration { HDL_Variable_or_Net_Declaration }

HDL_Variable_or_Net_Declaration ::=
      net_declaration
    | reg_declaration
    | integer_declaration
```

Syntax 8-91—Structure declaration

The nonterminals list_of_variable_identifiers, net_declaration, reg_declaration, and integer_declaration are defined in the syntax for IEEE Std 1364.

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.

Annex A

(normative)

Syntax rule summary

The annex summarizes the syntax.

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

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

vunit (;

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

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

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

e) Square brackets enclose optional items unless it appears in boldface, in which case it stands for itself. For example:

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

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

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

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

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

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

g) A colon (:) in a production starts a line comment unless it appears in boldface, in which case it stands for itself.