IEEE Standard for Property Specification Language (PSL)

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

and the
IEEE Standards Association Corporate Advisory Group

1850™

IEEE
3 Park Avenue
New York, NY 10016-5997, USA

6 April 2010

IEEE Std 1850™-2010
(Revision of
IEEE Std1850-2005)

IEEE Std 1850™-2010
(Revision of
IEEE Std1850-2005)

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 25 March 2010
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. 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 1354 (Verilog®), IEEE Std 1666™ (SystemC®), and IEEE Std 1800™ (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

---

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

PDF: ISBN 978-0-7381-6255-3 STD96063
Print: ISBN 978-0-7381-6256-0 STDPD96063

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 or the soundness of any judgments contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims 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 IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims 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. IEEE Standards documents are supplied **"AS IS."**

The existence of an IEEE 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 IEEE 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, or every ten years for stabilization. When a document is more than five years old and has not been reaffirmed, or more than ten years old and has not been stabilized, 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 IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other 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. A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal interpretation of the IEEE.At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE. Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Recommendations to change the status of a stabilized standard should include a rationale as to why a revision or withdrawal is required.

Comments and recommendations on standards, and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board

445 Hoes Lane

Piscataway, NJ 08854

USA

Introduction

This introduction is not part of IEEE Std 1850-2010, IEEE Standard for Property Specification Language (PSL).

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 2010 refines IEEE Std 1850 PSL 2005 by providing extensions for improved verification IP reuse (e.g., the vpkg type of vunit) and interaction between the assertions and the simulation environment (local variables), and by addressing minor technical issues. The formal semantics were updated to reflect these changes.

Notice to users

Laws and regulations

Users of these documents should consult all applicable laws and regulations. Compliance with the provisions of this standard does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

This document is copyrighted by the IEEE. It is made available for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making this document available for use and adoption by public authorities and private users, the IEEE does not waive any rights in copyright to this document.

Updating of IEEE documents

Users of IEEE standards should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect. In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE Standards Association website at http://ieeexplore.ieee.org/xpl/standards.jsp, or contact the IEEE at the address listed previously.

For more information about the IEEE Standards Association or the IEEE standards development process, visit the IEEE-SA website at http://standards.ieee.org.

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 amendment 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 is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this amendment are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

At the time this standard was submitted to the IEEE-SA for approval, the IEEE P1850 PSL Working Group had the following membership:

Harry Foster, Accellera, *Chair*
Sitvanit Ruah, IBM, *Co-Chair*
N. S. Subramanian, Cadence, *Secretary*
Anne Lustig-Picus, IBM, *Editor*
Hanan Singer, IBM, *Editor*

Mohamed-Lyes Benalycherif, ST-Ericsson
Surrendra Dudani, Synopsys
Cindy Eisner, IBM
Dana Fisman, IBM
Sandeep Gupta, Cadence
Joseph Lu, Altera
Sami Maisnemi, Nokia
Erich Marschner, Cadence
Johan Mårtensson, Jasper DA
Avigail Orni, IBM
Dmitry Pidan, IBM
Tej Singh, Mentor Graphics
Richard Wallace, Northrop Grumman
Yaron Wolfsthal, IBM

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

Accelera
Cadence Design
IBM
Mentor Graphics
Synopsys

The working group gratefully acknowledges the contributions of the following organizations and participants. Without their assistance and dedication, the initial standard would not have been completed.

Organization	*Participant*
Accellera	Harry Foster, *Chair*
Altraverifica Ltd.	Adriana Maggiore
Cadence	Sandeep Gupta
	Makarand Joshi
	Erich Marschner
	N. S. Subramanian, *Secretary*
	Stephen Ward
IBM	Cindy Eisner
	Dana Fisman
	Anne Lustig-Picus, *Editor*
	Avigail Orni
	Dmitry Pidan
	Sitvanit Ruah, *Co-chair*
	Hanan Singer, *Editor*
	Yaron Wolfsthal
Infineon	Klaus Winkelmann
Intel	Alex Levin
Jasper DA	Johan Alfredsson
	Johan Mårtensson
Mentor Graphics	Stephen Bailey
	Andrew Seawright
	Tej Singh
Nokia	Jari Kalinainen
	Sami Maisnemi
Northrop Grumman	Richard Wallace
Novas Software, Inc.	Bassam Tabbara
NVidia	Joseph Lu
Phillips	Sylvain Boucher
ST Microelectronics	Mohamed-Lyes Benalycherif
	Andrea Fedeli
Sun Microsystems	Tom Thatcher
Synopsys	Surrendra Dudani

When the IEEE-SA Standards Board approved this standard on 25 March 2010, it had the following membership:

Robert M. Grow, *Chair*
Richard H. Hulett, *Vice Chair*
Steve M. Mills, *Past Chair*
Judith Gorman, *Secretary*

John Barr
Karen Bartleson
Victor Berman
Ted Burse
Richard DeBlasio
Andy Drozd
Mark Epstein
Alexander Gelman
Jim Hughes
Young Kyun Kim
Joseph L. Koepfinger*
John Kulick
David J. Law
Ted Olsen
Glenn Parsons
Ronald C. Petersen
Narayanan Ramachandran
Jon Walter Rosdahl
Sam Sciacca

*Member Emeritus

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

Howard L. Wolfman, TAB Representative
Michael Janezic, NIST Representative
Satish K. Aggarwal, NRC Representative

Michelle Turner
IEEE Standards Program Manager, Document Development

Michael Kipness
IEEE Standards Program Manager, Technical Program Development

Contents

IEEE Standard for
Property Specification Language (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®),[1] IEEE Std 1364™ (Verilog®), IEEE Std 1800™ (SystemVerilog®), and IEEE Std 1666™ (SystemC®), to facilitate a common specification and verification flow for multi-language and mixed-language designs.

This standard creates an updated IEEE standard based upon IEEE Std 1850-2005. The updated standard will refine IEEE standard, addressing errata, minor technical issues, and proposed extensions specifically related to property reuse and improved simulation usability.

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

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 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.

1.2.2 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.

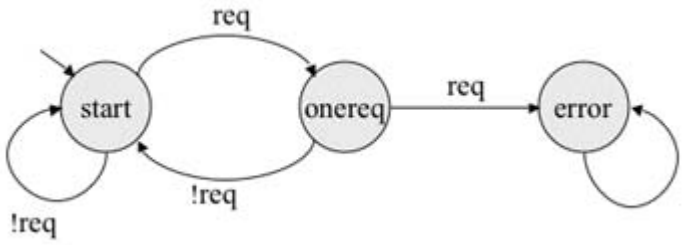

Figure 1—A simple (deterministic) FSM that checks Property 1

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.

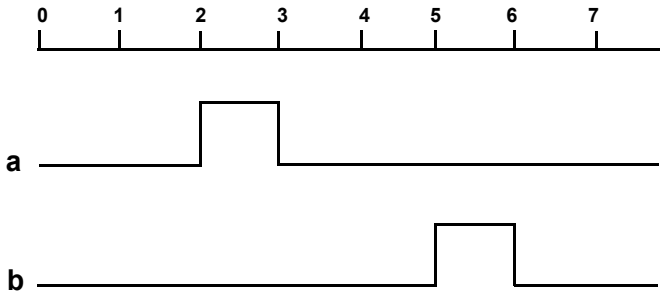

Figure 2—A trace that satisfies Property 2

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.

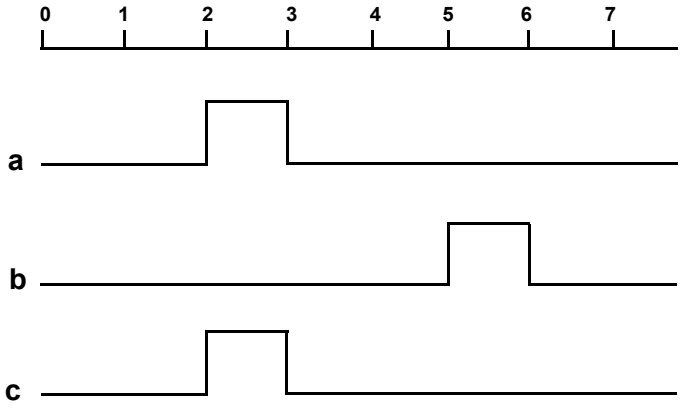

Figure 3—A trace that satisfies Property 3

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 (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated referenced, 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 Std 1666™, IEEE Standard for the SystemC Language.

IEEE Std 1800™, IEEE 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 document, the following terms and definitions apply. *The IEEE Standards Dictionary: Glossary of Terms & Definitions* should be referenced for terms not defined in this clause.[6]

3.1 Definitions

This subclause defines the terms used in this standard.

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

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.

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

behavior: A path.

Boolean (expression): An expression that yields a logical value.

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.

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

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.

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.

count: A number or range.

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.

cycle: An evaluation cycle.

describes: A term used to identify the set of behaviors for which Boolean expression, sequential expression, or property holds.

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.

design behavior: A computation path for a given design.

[6] *The IEEE Standards Dictionary: Glossary of Terms & Definitions* is available at http://shop.ieee.org/.

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.

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.

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

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

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*.

finite range: A range with a finite high bound.

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.

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.

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

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

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.

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.

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

model checking: A type of formal verification.

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

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

occurrence (of a Boolean expression): A cycle in which the Boolean expression holds.

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.

positive count: A positive number or a positive range.

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

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

prefix (of a given path): A path of which the given path is an extension.

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.

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.

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.

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.

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

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

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

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

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

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

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

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

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

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

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*.

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

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 Std 1800). The SystemVerilog flavor also has limited influence on the syntax of the temporal layer. For example, ranges of the temporal layer are specified using the SystemVerilog-style syntax `i:j`.

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 IEC/IEEE 62142). 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 Std 1666). 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 subclause 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!**	**rose**
AF	**EF**	**next_a**	
AG	**EG**	**next_a!**	**sequence**
AX	**EX**	**next_e**	**stable**
abort	**ended**	**next_e!**	**string**
always	**eventually!**	**next_event**	**strong**
and[a]		**next_event!**	**sync_abort**
assert	**F**	**next_event_a**	
assume	**fairness**	**next_event_a!**	**to**[e]
async_abort	**fell**	**next_event_e**	
	for	**next_event_e!**	**U**
before	**forall**	**nondet**	**union**
before!		**nondet_vector**	**until**
before!_	**G**	**not**[c]	**until!**
before_		**numeric**	**until!_**
bit	**hdltype**		**until_**
bitvector		**onehot**	
boolean	**in**	**onehot0**	**vmode**
	inf	**or**[d]	**vpkg**
clock	**inherit**		**vprop**
const	**is**[b]	**property**	**vunit**
countones	**isunknown**	**prev**	
cover			**W**
	mutable	**report**	**within**
default		**restrict**	
	never	**restrict!**	**X**
	next		**X!**

[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!_`.[7]

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.

[7]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 as follows:

`[* ]`	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 as follows:

`&`	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 as follows:

`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 as follows:

`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 as follows:

`|->`	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 as follows:

`->`	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 as follows:

`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 *(lowest precedence)*	right	`->` `<->`

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 | **vpkg** | **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, HDL_STMT, and HDL_SEQ_STMT connect the PSL syntax with the syntax for declarations and statements in the grammar for each HDL. All 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

Flavor Macro HDL_SEQ_STMT =
    SystemVerilog: SystemVerilog_statement_item
    / Verilog: Verilog_statement
    / VHDL: VHDL_sequential_statement
    / SystemC: SystemC_statement
    / GDL: GDL_process_item
```

Syntax 4-3—Flavor macros HDL_DECL, HDL_STMT, and HDL_SEQ_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 subexpressions 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 6.5.1. 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.

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 5-1. In such a case, there is no interpretation of type or values involved.

```
Any_Type ::=
  HDL_or_PSL_Expression
```

Syntax 5-1—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 5-2.)

```
Bit ::=
  bit_HDL_or_PSL_Expression
```

Syntax 5-2—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 5-3.)

```
Boolean ::=
  boolean_HDL_or_PSL_Expression
```

Syntax 5-3—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 5-4.

```
BitVector ::=
  bitvector_HDL_or_PSL_Expression
```

Syntax 5-4—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 Bit-Vector 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 5-5.

```
Number ::=
  numeric_HDL_or_PSL_Expression
```

Syntax 5-5—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 evaluatable—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 5-6.

```
String ::=
   string_HDL_or_PSL_Expression
```

Syntax 5-6—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 5-7 illustrates.

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

Syntax 5-7—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 5-8.)


```
HDL_Expression ::=
   HDL_EXPR

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

Syntax 5-8—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.

The meaning of the HDL expression is determined with respect to a given verification unit that acts as the root of a verification run—the root verification unit. The meaning of the HDL expression is consistent through all vunits inherited by the root verification unit.

NOTE—An HDL expression declared in a certain verification unit may have distinct meaning when it is the root vunit, and when it is inherited by another root vunit. It may also have distinct meaning when it is inherited by different root vunits.

A verification unit is said to transitively inherit a name or operator symbol if there exists a finite number of verification units V_1, V_2, ... ,V_{k-1}, V_k such that the following conditions are met:

a) V_1 is the given verification unit, and
b) for every i such that 1 <= i < k–1 it holds that V_i inherits using the default (transitive) **inherit** keyword V_{i+1}, and
c) V_{k-1} inherits V_k using either the transitive or nontransitive **inherit** keyword, and
d) V_k declares that name or operator symbol.

That is, a given verification unit V_1 transitively inherits a name or operator symbol if there exists a (possibly empty) path in the inheritance graph (see 7.2.3) to another verification unit V_k such that all edges except maybe the last are solid and the name or operator is declared in V_k.

For each name and operator symbol in the HDL expression, the meaning of the name or operator symbol in a given root verification unit is determined as follows:

1) If this is an operator symbol that is predefined in the flavor of PSL used, then the operator symbol has its predefined meaning.
2) If the root 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.
3) Otherwise, if the root verification unit transitively inherits 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.
4) Otherwise, if the root verification unit transitively inherits more than one declaration of this name or operator symbol, but all declarations appear in vunits that are related to each other by the transitive

closure with respect to inheritance, then the object created by the declaration closest to the root with respect to the inheritance relation is the meaning of this name or operator symbol.

5) 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.
6) 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.
7) 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 root verification unit [in step 1) or step 2)], or in the transitive closure of all inherited verification units where one is not related by inheritance to the other [in step 3) and step 4)], or in the default verification mode [in step 5)], or if the name is ambiguous at the end of the associated design module or instance [in step 6)].

NOTE—Whenever the text above refers to a declaration, the declaration may in particular be an override declaration.

Example

```
vunit A {
  wire a;
  inherit B;
}

vunit B {
  wire b;
  nontransitive inherit C;
}

vunit C {
  wire c;
  inherit D;
}

vunit D {
  wire d;
  inherit E;
}

vunit E {
  wire e;
}
```

The names a, b, and c (but not d and e) have meaning in verification unit A. In verification unit B, only names b and c have meaning. In verification unit C, only names c and d and e have meaning.

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 5-9.

```
HDL_or_PSL_Expression ::=
   PSL_Expression

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

Syntax 5-9—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 5-10.

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`.

```
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_Set )
    | nondet_vector ( Number, Value_Set)

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

Value_Range ::=
    Value
  | finite_Range

finite_Range ::=
    Low_Bound RANGE_SYM High_Bound
```

Syntax 5-10—Built-in functions

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 following timing diagram, 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 following timing diagram, 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()` built-in function. 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 following timing diagram, 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 following timing diagram, 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 following timing diagram, 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 evaluatable 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 shall 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()` shall 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 5-11, either of which can be the value of the resulting expression.

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

Syntax 5-11—Union expression

Restrictions

The two operands shall 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 5-12).

```
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 5-12—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 shall 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 as follows:

— (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 as follows:

— 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 shall 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 shall be equivalent to True.

5.4 Default clock declaration

A *default clock declaration*, shown in Syntax 5-13, 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 5-13—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 the following 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 Chapter 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), i.e., behaviors of the design. A sequential expression is said to hold tightly on a given finite path (see , 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 , 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) 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) in a cycle if it holds in that cycle. An occurrence of a Boolean expression (see) 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 6-1, describe single- or multi-cycle behavior built from a series of Boolean expressions.

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

Syntax 6-1—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 6-2, constructs a SERE that is the concatenation of two other SEREs.


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

Syntax 6-2—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 6-3, 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 6-3.)

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

Syntax 6-3—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 6-4.)

```
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 6-4—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 6-5, constructs a Compound SERE in which one of two alternative Compound SEREs hold at the current cycle.

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

Syntax 6-5—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 6-6, 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 6-6—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 6-7, 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 6-7—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 6-8, 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 6-8—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 6-9, 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 6-9—Parameterized SERE

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 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 the following our 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 shall 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 shall 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 2—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 $(k - j + 1) \times (n - m + 1)$ 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 6-10.)

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

Syntax 6-10—Sequences

Sequence Instances are described in 6.5.3.1. 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 6-11, constructs repeated consecutive concatenation of a given Boolean or Sequence.

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

Count ::=
   Number
  | Range

Range ::=
   Low_Bound RANGE_SYM High_Bound

Low_Bound ::=
   Number
  | MIN_VAL

High_Bound ::=
   Number
  | MAX_VAL
```

Syntax 6-11—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 6-12, 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 6-12—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 6-13, 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 6-13—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 6-14.[8]

```
Braced_SERE ::=
    {[ [[ HDL_DECL {HDL_DECL} ]] ] SERE }
| {[free( HDL_Identifier {,HDL_Identifier} )] SERE }
```

Syntax 6-14—Braced SERE

[8]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.

6.1.2.5 Clocked SERE (@)

The *SERE clock* operator (@), shown in Syntax 6-15, provides a way to clock a SERE.

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

Syntax 6-15—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.

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.

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 6-16, describe single- or multi-cycle behavior built from Boolean expressions, sequential expressions, and subordinate properties.

```
FL_Property ::=
   Boolean
| ([ [| HDL_DECL {,HDL_DECL} |] ] FL_Property )
```

Syntax 6-16—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 that specify that the behavior described by a sequence occurs. (See Syntax 6-17.)

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

Syntax 6-17—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 any finite prefix of the given path.

NOTE—If S contains no contradictions, a simpler 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 every finite prefix of 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 6-18, provides a way to clock an FL Property.

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

Syntax 6-18—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 6-19, specifies that an FL Property holds at all times, starting from the present.

FL_Property ::=
always FL_Property

Syntax 6-19—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 6-20, specifies that an FL Property or a sequence never holds.

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

Syntax 6-20—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 6-21, specifies that an FL Property holds at the current cycle or at some future cycle.

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

Syntax 6-21—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 6-22, 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 6-22—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 6-23, 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 6-23—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 6-24, 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 6-24—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 6-25, 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 6-25—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 6-26, 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 6-26—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 6-27, 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 6-27—next_event_e operators

The rightmost operand of the `next_event_e!` or `next_event_e` operator is a 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 6-28, 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 6-28—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 6-29, 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 6-29—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 6-30, 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 6-30—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 6-31, 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 6-31—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 6-32, 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 6-32—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, as follows:

— 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 shall 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 shall 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 6-33, is used to specify logical implication.

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

Syntax 6-33—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 6-34, is used to specify the iff (if and only if) relation between two properties.

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

Syntax 6-34—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 6-35, is used to specify logical and.

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

Syntax 6-35—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 6-36, is used to specify logical or.

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

Syntax 6-36—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 6-37, is used to specify logical negation.


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

Syntax 6-37—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 6-38, 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 6-38—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 6-39, 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 6-39—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 6-40, specifies that an OBE property holds at all next states of the given state.

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

Syntax 6-40—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 6-41, specifies that an OBE property holds at the given state and at all future states.

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

Syntax 6-41—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 6-42, 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 6-42—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 6-43, 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 6-43—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 6-44, 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 6-44—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 6-45, 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 6-45—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 6-46, 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 6-46—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 6-47, 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 6-47—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 6-48, is used to specify logical implication.

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

Syntax 6-48—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 6-49, is used to specify the *iff* (if and only if) relation between two properties.

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

Syntax 6-49—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 6-50, is used to specify logical and.

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

Syntax 6-50—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 6-51, is used to specify logical or.

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

Syntax 6-51—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 6-52, is used to specify logical negation.

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

Syntax 6-52—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 6-53. 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: )
```

Syntax 6-53—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.

- 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.
- 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, as follows:

- 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 shall 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 shall 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 Local variables

A local variable is declared inside a property enclosed in parentheses or inside a braced sequence. The local variable declaration uses the syntax of the underlying flavor language and is enclosed within the delimiters "[[" and "]]".

Local variables can be of any type supported by the underlying flavor language.

The variables can be modified inside procedural blocks using the underlying flavor language syntax (see 6.4). Syntax 6-54 shows the syntax for procedural blocks. Syntax 6-55 shows the syntax for SEREs and properties with local variable declaration, and the syntax for the "free" operator. The free operator when applied to a local variable removes the variable from the current scope. Syntax 6-56 shows the syntax for attaching a procedural block to a Boolean or sequence.

```
Proc_Block ::=
   [| Proc_Block_Item { Proc_Block_Item }|]
Proc_Block_Item::=
   HDL_DECL
   | HDL_SEQ_STMT
```

Syntax 6-54—Procedural block

```
Braced_SERE ::=
   { [ [| HDL_DECL {HDL_DECL} |] ] SERE }
   | { [free ( HDL_Identifier {, HDL_Identifier} )] SERE }
FL_Property ::=
   ([ [| HDL_DECL {HDL_DECL} |] ] FL_Property )
```

Syntax 6-55—Local variable declaration

```
Sequence ::=
   Sequence Proc_Block
Repeated_SERE ::=
   Boolean Proc_Block
   | Sequence Proc_Block
SERE ::=
   Boolean Proc_Block
```

Syntax 6-56—Sequence/Boolean with procedural blocks

A local variable is visible in the SERE or property in which it is declared and in any sub-SERE or sub-property of the SERE or property in which it is declared, except if it has been freed using the free operator. A local variable is visible in a procedural block if it is visible in the Boolean expression to which the procedural block is attached. A local variable can be referenced by any Boolean expression or procedural block in which it is visible. A local variable can be modified by any procedural block in which it is visible.

A local variable keeps its value until the value is changed by an assignment or until the local variable is freed with the free operator.

Example 1

Consider the property

```
( [[reg [31:0] i <= 32'd0; ]]{a; b [[i <= i+32'd1; ]];
  c ; d [[i <= i+32'd1; ]]} |=> {e})
```

Local variable i is initialized to 0. At the point where a occurs it has the value 0. At the point where b occurs it also has the value 0, and is then incremented to 1. At the point where c occurs it has the value 1, and at the point where d occurs it has the value 1 and is then incremented to 2. At the point where e occurs, it has the value 2.

A single local variable declaration may result in multiple copies of a local variable as the result of a temporal operator or a SERE repetition operator.

Example 2

If we enclose the property in Example 1 in an **always** operator, like this:

```
always ( [[reg [31:0] i <= 32'd0; ]]
        { a ; b [[i <= i+32'd1; ]] ;
          c ; d [[i <= i+32'd1; ]]} |=> {e})
```

then there are multiple independent copies of local variable i, one per cycle, because the declaration itself is enclosed in the always operator.

Example 3

If a local variable declaration appears in a SERE, like this:

```
{ [[reg [31:0] i<=32'd0; ]] a ; {b [[i <= i+32'd1;]] | c} ; d} |=> {e}
```

then adding a repetition operator such as the following:

```
{[*] ; { [[reg [31:0] i<=32'd0; ]] a ;
{b [[i <= i+32'd1;]] | c} ; d}} |=> {e}
```

results in multiple independent copies of local variable i, because the declaration itself is reached at every cycle (because of the [*]).

Even when the declaration is not modified by an **always** operator or a SERE repetition operator, there may be multiple independent copies of a local variable as a result of multiple matches of a SERE.

Example 4

Consider the following SERE:

```
{ [[reg [31:0] i<=32'd0; ]]a[*]; b[[i <= i+32'd1; ]][*];c} |=> {d}
```

There is a single declaration of `i`, but if the left-hand SERE matches the path in multiple ways, then multiple independent copies of the local variable will be born. For example, see the following trace in Figure 4.

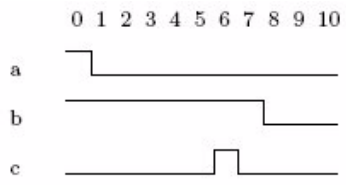

Figure 4—Example 4

The left-hand SERE holds tightly on cycles 0–6 and it does so in multiple ways. If we have matched zero `a`'s with `a[*]`, then the value of `i` at evaluation cycle 6 is 6, and if we have matched one `a` with `a[*]`, then the value of `i` at evaluation cycle 6 is 5.

NOTE—We use the terms "multiple independent copies" and "will be born" figuratively. A tool is free to implement local variables and keep track of the values it may have in ways that do not actually spawn multiple copies of a local variable (for example, using a non-deterministic automaton, a sparse array or other efficient data structure) as long as the behavior of the local variables as seen by the user is consistent with the behavior that would be exhibited by multiple independent copies.

Example 5

```
([[reg [31:0] count;]]
{[*]; start [[count<=32'd0 ;]];
{error[->][[count<=count+32'd1;]]}[*];end} |->
{count <= MAX_ERROR})
```

The property above passes if the number of errors between `start` and `end` does not exceed `MAX_ERROR`. The property declares the local variable `count` of type `reg[31:0]` that counts the number of errors. This is done by initializing `count` to 0 when `start` holds and incrementing it by 1 whenever `error` holds.

Finally, `count` is compared to `MAX_ERROR` when `end` holds. Note that if there are overlapping sequences of cycles starting with `start` and ending with `end`, then there will be multiple independent copies of the local variable `count`.

For example, see the trace in Figure 5.

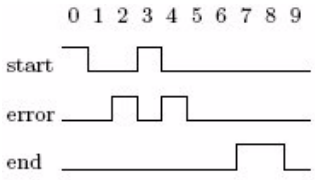

Figure 5—Example 5

At cycle 0 there is only one copy of the local variable `count` corresponding to the `start` of cycle 0. At cycle 3 another copy of the local variable `count` is born which counts the number of errors between the `start` of cycle 3 and `end`. At cycle 4, the first copy has value 2 and the second copy has value 1.

Example 6

Consider the following property:

```
always ([[ reg [31:0] count_r<= 32'd0; reg [31:0] count_w<= 32'd0; ]]
  {{fifo_empty;
    { free(count_w) read_req[->][[count_r<=count_r+32'd1;]][*];
      !read_req[*]} &&
    { free(count_r) write_req[->][[count_w<=count_w+32'd1;]][*];
      !write_req[*]};
    fifo_empty }} |-> {count_r==count_w})
```

The property holds if between any two cycles in which the FIFO is empty the number of read requests is equal to the number of write requests. The property declares the local variables `count_r` and `count_w`. The left hand side operand of the **&&** operator makes the local variable `count_r` count the number of read requests. The right hand side operand counts the number of write requests. Each of the operands "frees" the local variable it does not assign in order to let it take on any value. At the end of the sequence of cycles matched by the left-hand side of the |- > (in which the FIFO is once again empty), `count_r` and `count_w` are compared to check if they are equal. Since `count_r` and `count_w` are referred to outside the scope of the **&&** operator, they have to be declared outside the **&&**.

Example 7

The following example shows what would happen in the case that the **free** operator is not used to free `count_w` and `count_r` as in the previous example. Since the left operand increments `count_r` but the right operand does not (and vice versa for `count_w`), we get that the left-hand side of the suffix implication does not hold tightly on any sequence of cycles. Thus, the property passes vacuously on every path.

```
always ([[ reg [31:0] count_r<= 32'd0; reg [31:0] count_w<= 32'd0; ]]
{{fifo_empty;
{ read_req[->][[count_r<=count_r+32'd1;]][*];!read_req[*]} &&
{ write_req[->][[count_w<=count_w+32'd1;]][*];!write_req[*]};
fifo_empty }} |-> {count_r==count_w})
```

6.4 Procedural blocks

The goal of a procedural block is modifying local variables and printing output through the evaluation of a property or SERE. A procedural block is triggered whenever we have seen a finite prefix of a path on which the property holds weakly, and the last cycle of the path corresponds to the location in the property at which the procedural block is attached. In fusion and overlapping suffix implication operators procedural blocks that are attached to the end of the left operand are triggered whenever the left operand is matched regardless of the right operand, and the evaluation of the right operand starts after the completion of the procedural blocks in the left operand.

Example 8

Consider the property and timing diagram as follows:

```
([[reg [31:0] i<= 32'd0;]]{[*];
{a;b}[[i<=32'd3]] : {c[[i<=i+32'd1;d}[[$display("%d",i)]]}
|->{e;f})
```

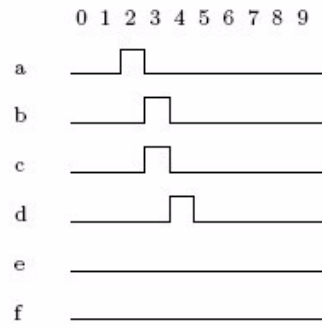

Figure 6—Example 8

The value of `i` at cycle 3 is 4 because the procedural block attached to `b` is triggered before the procedural block attached to `c`. The value of `i` (4) is displayed at cycle 4 because `d` holds on this cycle and the fact that `e` does not hold at this cycle does not affect the triggering.

All variables visible to a property or SERE are automatically visible in procedural blocks inside this property/SERE.

Restrictions

- Procedural blocks can be attached only to maximal Booleans (that is, Booleans that are not part of another Boolean) or to sequences.
- The code inside a procedural block should be such that it can be inserted into a procedure or function "shell" (i.e., an empty procedure or function) and be syntactically legal in the flavor language. The local variables visible in the Boolean or sequence to which the procedural block is attached are regarded as being implicitly passed by reference to the procedure.
- A procedural block cannot consume time. It should complete execution in atomic time as a basic block. In particular wait statements and their equivalents (e.g., # of delays in Verilog) are not allowed inside procedural blocks.
- A procedural block contains only sequential statements. In particular, fork/join constructs are not allowed.

— Procedural blocks shall be triggered only when the property in which they are declared is used as an assertion or cover. It should not be triggered when used as an assumption or in a fairness or restrict verification directive.
— Procedural blocks should not change the environment or the design.

Example 9

Consider the following property:

```
{[[reg [31:0] lv;]] [*];a[[lv<=32'd0;]]; b[[lv<=lv+32'd1;]][*];
c[[$display("%d",lv)]]}
```

Denote the procedural blocks in the SERE above by the following:

```
A1 : lv<= 32'd0;
A2 : lv<=lv + 32'd1;
A3 : $display("%d", lv);
```

The timing diagram shown in Figure 7 has two matches for the SERE above.

The timing diagrams in Figure 8 and Figure 9 show these matches and the triggering of procedural blocks in each of these matches.

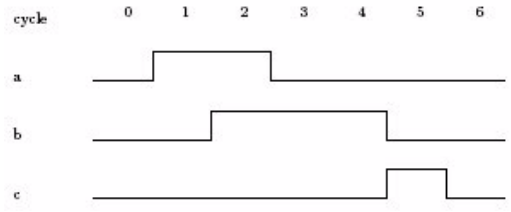

Figure 7—Example 9

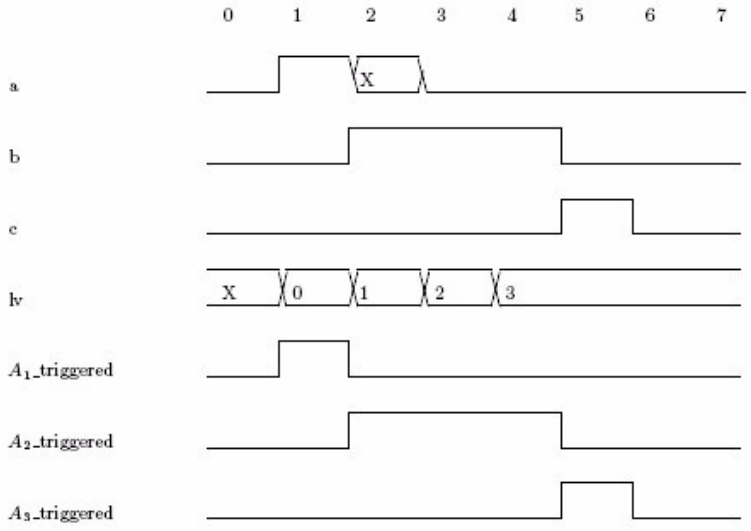

Figure 8—Match 1 of the SERE of Example 9

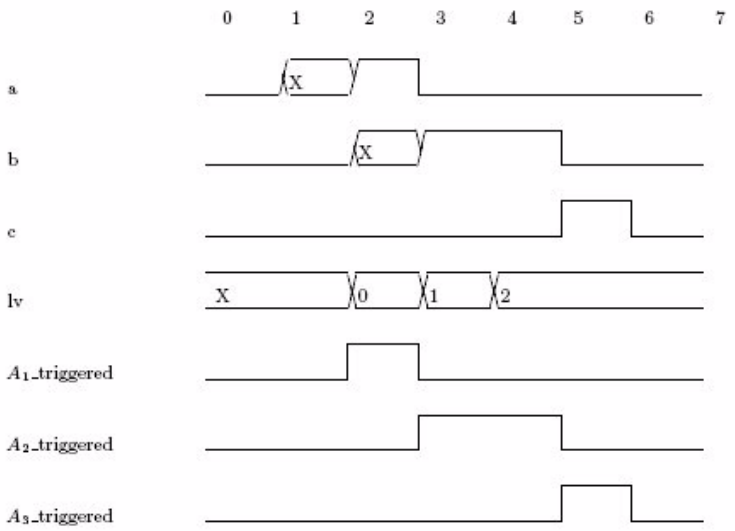

Figure 9—Match 2 of the SERE of Example 9

Figure 10 shows the timing diagram of Figure 7 with the triggering of procedural blocks A_1, A_2, and A_3, that is, it shows the union of the triggerings of match 1 and match 2. The order of triggering of procedural blocks corresponding to different matches is arbitrary, i.e., if more than one procedural block is triggered at the same cycle (for different matches), there is no way to predict in which order they will be executed. For example, in Example 6 both A_1 and A_2 are triggered on cycle 2 for different matches. There is no way to predict in what order they will be triggered. Note, however, that since procedural blocks update only local variables, and every procedural block updates its own copy of the local variable the order of triggering will only affect the order of reporting.

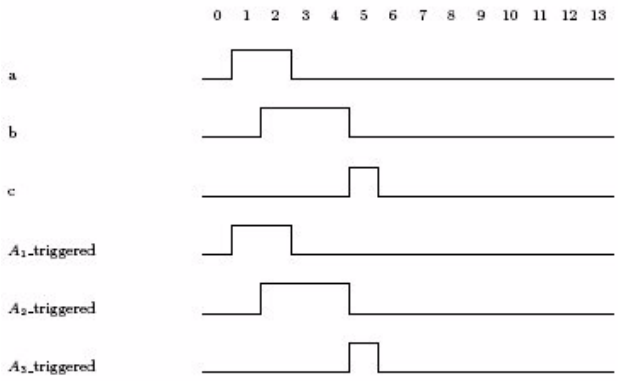

Figure 10—Example 9: Triggering of procedural blocks

Example 10

Consider the following property:

```
([[ reg[31:0] lv; ]]
 {[*]; a[[lv<=32'd0;]];
```

```
{{b[[lv<=lv+32'd1;]][+]} | {c[[lv<=lv+32'd1;]][+]; d}} }
|=>{e [[$display("%d", lv);]]})
```

Denote the procedural blocks in the previous SERE by

A_1 = lv <= 32'd0;
A_2 = lv <= lv + 32'd1;
A_3 = lv <= lv + 32'd1;
A_4 = $display("%d", lv);

The way to understand triggering of procedural blocks in a suffix implication property is to consider the SERE formed by concatenating the left and right sides of the suffix implication operator. In the example above, the resulting SERE is as follows:

```
({[*]; a[[lv<=32'd0;]];
 {{b[[lv<=lv+32'd1;]][+]} | {c[[lv<=lv+32'd1;]][+]; d}};
 e [[$display("%d", lv);]]})
```

SERE 1

The timing diagram shown in Figure 11 has three matches of the above SERE. The SERE holds tightly on paths on which the property holds non-trivially (that is, the left hand side holds tightly). These matches are shown in Figure 12 through Figure 14.

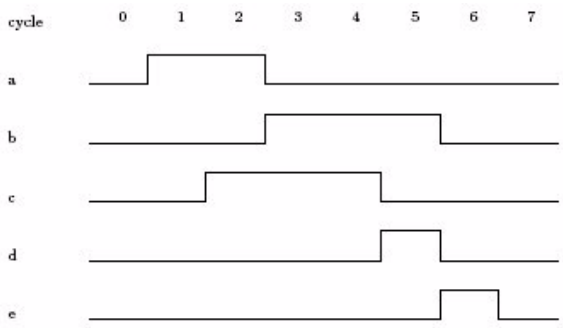

Figure 11—Example 10

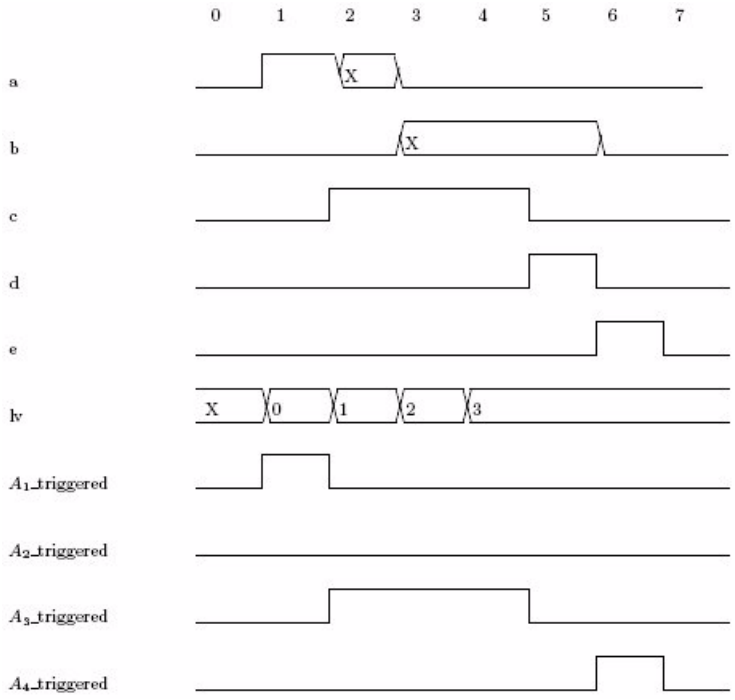

Figure 12—Match 1 of SERE 1

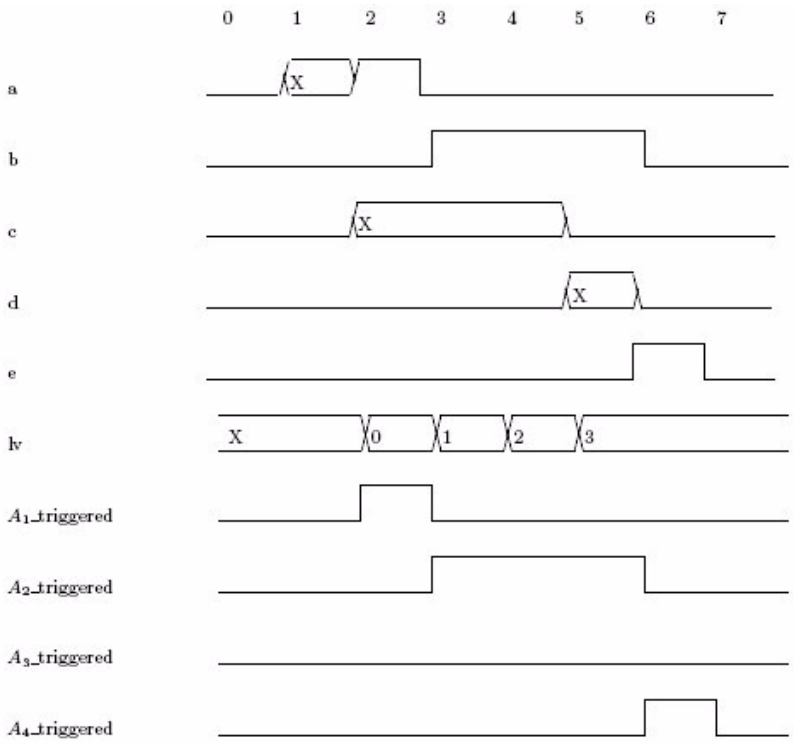

Figure 13—Match 2 of SERE 1

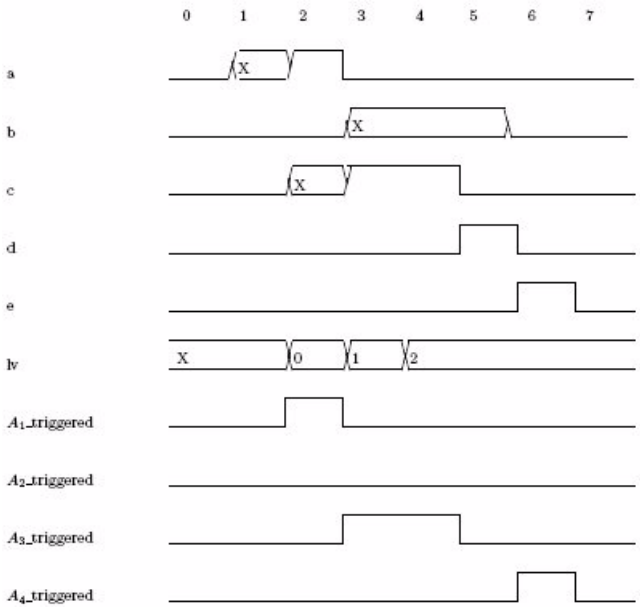

Figure 14—Match 3 of SERE 1

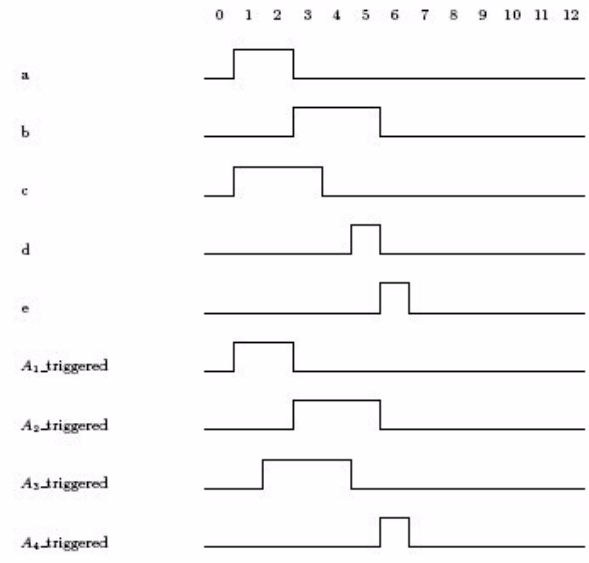

Figure 15—Triggering of procedural blocks in the path in Figure 11

Figure 15 shows the timing diagram of Figure 11 with the triggering of procedural blocks A_1, A_2, A_3, and A_4; that is, it shows the union of the triggerings of all matches.

A_1 is triggered at time 1 because match 1 reaches its time point and at time 2 because matches 2 and 3 reach its time point. A_2 is triggered at times 3, 4, and 5 because match 2 reaches its time point. A_3 is triggered at times 2, 3, and 4 because matches 1 and 3 reached its time point. A_4 is triggered at time 6 since all the three matches reached its time point.

6.5 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 6-57.)

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.

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 | mutable] 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 6-57—Property declaration

Restrictions

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

NOTE 1—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.

NOTE 2—Writing **const** without a Value_Parameter is the same as writing **const numeric** and is left in for backward compatibility.

6.5.1 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.

6.5.1.1 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 (see Table 5).

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.

6.5.1.2 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 shall 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);
```

Named properties and sequences can contain procedural blocks. The formal parameters of the named constructs can be refered to inside these procedural blocks. A formal parameter that is a Value Parameter can optionally be qualified with mutable. Named constructs that contain mutable formal parameters can be instantiated only within sequences or properties. Formal parameters that are qualified with mutable can be changed inside procedural blocks in the named construct. Since procedural blocks can change only local variables, the actual expression that maps to a mutable formal parameter shall be a local variable visible in the instantiating sequence or property. If the instantiated named construct modifies the value of the mutable formal parameter, the instantiating sequence or property sees the modified value of the local variable that maps to this mutable formal parameter.

The syntax of named constructs parameters is shown in Syntax 6-58.

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

Syntax 6-58—Syntax of named constructs parameters

Example 8

Sequence count cycles are defined as follows:

```
sequence count_cycles(boolean sig;
                      mutable hdltype reg [31:0] num_cycles)=
      {a[[num_cycles<=num_cycles+32'd1]][*]};

assert always
  ([[ reg [31:0] num_a<=32'd0; reg [31:0] num_b<=32'd0; ]]
     {start; count_cycles(a, num_a);
      count_cycles(b, num_b); end }
    |-> {num_a==num_b});
```

The property above checks that whenever we have a sequence on which `{start; a[*]; b[*]; end}` holds tightly, the number of `a` and `b` occurences between `start` and `end` are equal. The property declares the local variables `num_a` and `num_b` and initializes them to 0. `num_a` is passed as **mutable** to the first instance of `count_cycles`, which counts the number of occurences of `a`. Similarly, the second instance of `count_cycles` counts the number of `b` occurences into the local variable `num_b`. Finally `num_a` and `num_b` are compared for equality.

6.5.2 Declarations

6.5.2.1 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 numeric 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.

6.5.2.2 Property declaration

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

Example

```
property ResultAfterN
    (boolean start; property result; const numeric 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 numeric 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.

6.5.3 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 6.5.1. 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.

6.5.3.1 Sequence instantiation

A sequence instantiation, shown in Syntax 6-59, 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 6-59—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 6.5.2.1,

```
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 }
```

6.5.3.2 Property instantiation

A property instantiation, shown in Syntax 6-60, 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 6-60—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 6.5.2.2,

```
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 (see Table 7-1).

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

Verification_Directive ::=
   Assert_Directive
 | Assume_Directive
 | Restrict_Directive
 | Restrict!_Directive
 | Cover_Directive
 | Fairness_Directive
```

Syntax 7-1—Verification directives

A verification directive may be preceded by a label. If a label is present, it shall 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 (see Table 7-2). Such unique names can be used by a tool for selective control and reporting of results.

```
Label ::=
   PSL_Identifier
```

Syntax 7-2—Labels

NOTE 1—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.

NOTE 2—The directives **assume_guarantee** and **restrict_guarantee** are no longer in the language.

7.1.1 assert

The Assert Directive, shown in Syntax 7-3, instructs the verification tool to verify that a property holds.

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

Syntax 7-3—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 following message:

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

7.1.2 assume

The Assume Directive, shown in Syntax 7-4, instructs the verification tool to constrain the verification (e.g., the behavior of the input signals) so that the given property holds.

Assume_Directive ::=
assume Property **;**

Syntax 7-4—Assume Directive

Restrictions

The Property that is the operand of an Assume Directive shall 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 1—Verification tools are not obligated to verify the assumed property.

NOTE 2—See 7.1.4 for additional examples related to the Assume Directive, as well as a comparison to other directives.

7.1.3 restrict

The Restrict Directive, shown in Syntax 7-5, is a special form of assumption, which uses a sequence instead of a property. Like the Assume Directive, the Restrict Directive instructs the verification tool to constrain the verification, e.g., by constraining the behavior of design inputs.

```
Restrict_Directive ::=
  restrict Sequence ;
```

Syntax 7-5—Restrict Directive

Informal Semantics

Let End-Of-Path (EOP) be a signal that is asserted on the last cycle of every finite path in the design (and is deasserted on all other cycles). For a given sequence S, the directive

```
restrict S ;
```

is equivalent to the directive

```
assume { S : EOP } ;
```

In other words, `restrict S;` instructs the verification tool to constrain the verification, so that the property

```
{ S : EOP } ;
```

holds on every path in the design.

Examples

See 7.1.4 for examples related to the Restrict Directive, as well as a comparison to other directives.

7.1.4 restrict!

The Restrict! Directive, shown in Syntax 7-6, is related to the Restrict Directive, and is also used for constraining the verification as specified by a given sequence.

```
Restrict!_Directive ::=
  restrict! Sequence ;
```

Syntax 7-6—Restrict! Directive

Informal Semantics

Let EOP (End-Of-Path) be a signal that is asserted on the last cycle of every finite path in the design (and is deasserted on all other cycles). For a given sequence S, the directive

```
restrict! S ;
```

is equivalent to the directive

```
assume { S : EOP } ! ;
```

In other words, `restrict! S;` instructs the verification tool to constrain the verification, so that every path in the design exactly matches the sequence S (i.e., every path models tightly S).

NOTE 1—When a Restrict! Directive is applied to the verification, all infinite paths are eliminated from the design, since an infinite path cannot tightly match a sequence.

NOTE 2—For a given sequence S, each of the following verification directives has a different meaning:

```
restrict S ;
restrict! S ;
assume S ;
assume S ! ;
```

Table 6 shows the differences between the four directives, when they are applied to the sequence `{!rst; rst[*3]}`.

Table 6—Differences between directives I

Directive	Description
`restrict {!rst; rst[*3]};`	*Informal semantics* The property `{{!rst;rst[*3]} : EOP}` holds on every path in the design *Explanation* Every path in the design is of length *at most* 4, and obeys the restriction that `rst` is low on the first cycle and high for each of the next 3 cycles (if they exist)
`restrict! {!rst; rst[*3]};`	*Informal semantics* The property `{{!rst;rst[*3]} : EOP}!` holds on every path in the design *Explanation* Every path in the design is of length *exactly* 4, and obeys the restriction that `rst` is low on the first cycle and high for each of the next 3 cycles
`assume {!rst; rst[*3]} ;`	*Informal semantics* The property `{!rst; rst[*3]}` holds on every path in the design *Explanation* Every path in the design may be of *any length*, and obeys the restriction that `rst` is low on the first cycle and high for each of the next 3 cycles (if they exist)
`assume {!rst; rst[*3]} ! ;`	*Informal semantics* The property `{!rst; rst[*3]}!` holds on every path in the design *Explanation* Every path in the design is of length *at least* 4, and obeys the restriction that `rst` is low on the first cycle and high for each of the next 3 cycles

Table 7 shows the differences between the four directives, when they are applied to the sequence {`!rst; rst[*3]; !rst[*]`}

Table 7—Differences between directives II

Directive	Description
`restrict {!rst; rst[*3]; !rst[*]};`	*Informal semantics* The property {{`!rst;rst[*3];!rst[*]`} : `EOP`} holds on every path in the design *Explanation* Every path in the design may be of *any length*, and obeys the restriction that `rst` is low on the first cycle, `rst` is high for the next 3 cycles (if they exist), and `rst` is low for all remaining cycles until the end of the path (if they exist)
`restrict! {!rst; rst[*3]; !rst[*]};`	*Informal semantics* The property {{`!rst;rst[*3];!rst[*]`} : `EOP`}`!` holds on every path in the design *Explanation* Every path in the design is *finite*, of length *at least* 4, and obeys the restriction that `rst` is low on the first cycle, `rst` is high for the next 3 cycles, and `rst` is low for all remaining cycles until the end of the path (if they exist)
`assume {!rst; rst[*3]; !rst[*]} ;`	*Informal semantics* The property {`!rst; rst[*3];!rst[*]`} holds on every path in the design *Explanation* Same as `assume {!rst; rst[*3]};`
`assume {!rst; rst[*3]; !rst[*]} ! ;`	*Informal semantics* The property {`!rst; rst[*3];!rst[*]`}`!` holds on every path in the design *Explanation* Same as `assume {!rst; rst[*3]} ! ;`

7.1.5 cover

The Cover Directive, shown in Syntax 7-7, 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 7-7—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.

7.1.6 fairness and strong_fairness

The Fairness Directives, shown in Syntax 7-8, are special kinds of assumptions that correspond to liveness properties.

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

Syntax 7-8—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 7-9, is used to group PSL declarations, directives, and modeling code.

```
Verification_Unit ::=
    Vunit_Type PSL_Identifier [ ( Context_Spec ) ] {
        { Inherit_Spec }
      { Override_Spec }
      { Vunit_Item }
}
Vunit_Type ::=
   vunit | vpkg | vprop | vmode
Context_Spec ::=
    Binding_Spec | Formal_Parameter_List

Binding_Spec ::=
   Hierarchical_HDL_Name

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 ::=
   [nontransitive] inherit vunit_Name { , vunit_Name } ;

Vunit_Item ::=
    HDL_DECL
    | HDL_STMT
    | PSL_Declaration
    | PSL_Directive
    | Vunit_Instance

Override_Spec ::=
     override Name_List ;

Name_List ::=
   Name { ',' Name }

Formal_Parameter_List ::=
Formal_Parameter { ; Formal_Parameter }
```

Syntax 7-9—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 7.2.3 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.

The default vmode (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.

Verification units can be used in various ways to specify a verification task. For example:

1) An abstract verification task can be modeled using a set of verification units, none of which are bound to a design.
2) A verification task related to a design can be modeled using a set of verification units, all of which are related to one another through inheritance or instantiation, and at least one of which is bound to a design module or instance.
3) A verification task related to a design can also be modeled using multiple sets of verification units, where each set is structured as defined in item 2), but each set is bound to possibly different modules or instances in the design.

A given tool may support any of these or other use models.

vunit

A verification unit which has the Vunit_Type as 'vunit' is called a vunit. A vunit is a verification unit intended for general purpose usage.

A vunit may contain any kind of Vunit Item. A vunit may also inherit or instantiate other verification units.

A vunit may be bound to a design module or instance, or instantiated, or inherited.

Restrictions

It is an error if a vunit has a binding_spec and is instantiated.

vpkg

A verification unit which has the Vunit_Type as 'vpkg' is called a vpkg. A vpkg is a verification unit specifically intended for encapsulating a set of declarations for reuse.

A vpkg may contain any PSL declaration. A vpkg may also inherit or instantiate other verification units.

A vpkg may be inherited or instantiated.

Restrictions

A vpkg shall not contain PSL directives.

A vpkg does not inherit the default vmode.

A vpkg shall not contain override statements.

A vpkg shall not be bound to an HDL module or instance. All signals or variables referenced within a vpkg shall be defined as parameters of either an individual declaration (local parameters) or of the vpkg itself (global parameters).

If a vpkg A inherits or instantiates a verification unit B, then B shall satisfy the restrictions that apply to a vpkg, regardless of whether B is declared as a vpkg or as some other kind of verification unit. This allows the user who wants to create a vpkg to use declarations defined in an existing vunit without having to change the vunit to a vpkg.

NOTES—A vpkg as a whole shall be parameterized (and instantiated), or the declarations contained within the vpkg shall be parameterized (and instantiated), or some combination of these two approaches shall be used in order to use the vpkg's declarations in a given context.

Example 1

```
// declaration of a parameterized vpkg
    vpkg Pkg1 (boolean req, ack; const T) {
        property ReqAckT = always ( req -> next {{ack} within [*T]} );
       property AckBeforeReq = always ( req -> next (ack before req) );
    }
    ...
    // instantiation of this vpkg
    P1: vpkg Pkg1 (rq, ak, 5);
    ...
    // use of this vpkg's declarations
    assert  P1.ReqAckT;
    assume  P1.AckBeforeReq;
    ...
```

Example 2

```
// declaration of a vpkg containing parameterized declarations
    vpkg Pkg2 {
        property ReqAckT (boolean req, ack; const T) =
             always ( req -> next {{ack} within [*T]} );
        property AckBeforeReq (boolean req, ack) =
             always ( req ->( next ack before req) );
    }
    ...
    // instantiation of this vpkg
    P2: vpkg Pkg2;
    ...
    // use of this vpkg's declarations
```

```
assert  P2.ReqAckT(rq,ak,5);
assume  P2.AckBeforeReq(rq,ak);
...
```

Example 3

```
// declaration of a partially parameterized vpkg containing
//     parameterized declarations
    vpkg Pkg3 (boolean ack) {
        property ReqAckT (boolean req, const T) =
         always ( req -> next {{ack} within [*T]} );
    }
    ...
    // instantiation of this vpkg
    P3: vpkg Pkg3 (ak); // all requests get a common acknowledge (ak)
    ...
    // use of this vpkg's declarations
    assert  P3.ReqAckT(req1, 5);     // higher priority request
    assert  P3.ReqAckT(req2, 10);   // lower priority request
    ...
```

vmode

A verification unit that has the Vunit_Type as 'vmode' is called a vmode. A vmode is a verification unit specifically intended for specifying a verification environment.

A vmode can contain modeling code, PSL declarations, and PSL assume, restrict, and fairness directives. A vmode can also contain instantiations of other verification units. A vmode may also inherit other verification units.

A vmode that is 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 default verification unit may not inherit other verification units of any type.

A vmode may be bound to a design module or instance, or instantiated, or inherited.

A vmode shall not contain an assert directive or a cover directive.

If a vmode A inherits or instantiates a verification unit B, then B shall satisfy the restrictions that apply to a vmode, regardless of whether B is declared as a vmode or as some other kind of verification unit.

vprop

A verification unit that has Vunit_Type as 'vprop' is called a vprop. A vprop is a verification unit specifically intended for specifying properties to verify.

A vprop can contain modeling code and PSL declarations. It may contain only assert and cover PSL directives. A vprop can also contain instantiations of other verification units. A vprop may also inherit other verification units.

A vprop may be bound to a design module or instance, or instantiated, or inherited.

Restrictions

If a vprop A inherits or instantiates a verification unit B, then B shall satisfy the restrictions that apply to a vprop, regardless of whether B is declared as a vprop or as some other kind of verification unit.

A vprop may not override design signals.

7.2.1 Verification unit binding

A verification unit may be bound to a design module or instance. Binding allows a verification unit to reference the names visible in that design module or instance.

Binding does not affect the visibility of names in locations other than the bound verification unit. In particular, if verification units A and B are both bound to design module or instance M, and neither A nor B inherit the other, then the names declared in A are not visible in B, and the names declared in B are not visible in A.

A verification unit A may be bound to a design module or instance M, regardless of whether the flavor of A and the implementation language of M are the same or are different. If they are different, then implicit cross-language type conversions are performed as required, following the conventions for mixed-language simulation.

A vunit with no binding_spec is considered to be unbound.

The declarations in an unbound vunit shall be bound using the binding rules of the inheriting or instantiating vunit if the unbound vunit is inherited or instantiated in a bound vunit.

Restrictions

A parameterized verification unit shall not be bound.

A vpkg shall not be bound.

An unbound verification unit may only inherit another unbound verification unit.

7.2.2 Verification unit instantiation

A verification unit instantiation, shown in Syntax 7-10, creates an instance of a verification unit. An instance of a verification unit is also a verification unit.

```
Vunit_Instance ::=
        Label ':'  Vunit_Type /vunit/_Name [ '('
Actual_Parameter_List ')' ] ';'
```

Syntax 7-10—Verification unit instantiation

The Vunit_Type specified in the Vunit Instance shall agree with that of the verification unit with the specified name.

Informal Semantics

Each instantiation of a verification unit V within another verification unit V2 creates a unique copy of the instantiated verification unit V, accessible only within verification unit V2.

A verification unit developed in a particular flavor of PSL may instantiate a verification unit developed in a different flavor of PSL. In this case, implicit cross-language type conversions are performed as required, following the conventions for mixed-language simulation.

Examples

```
vunit V1 (logic x, y ){
…
…
}

vunit V2 (top.i1)  {
// top.i1 is a verilog model
…
        V1_inst: vunit V1(D, reset);

}
```

NOTE—The instantiation rules of 6.5.3 apply to vunit instances as well.

7.2.3 Verification unit inheritance

One verification unit may inherit another verification unit. Inheritance allows the inheriting verification unit to reference the declarations contained in the inherited verification unit. Inheritance can also be used as one means of composing a set of directives that, in aggregate, define a verification task.

Inheritance does not affect the visibility of names in locations other than the inheriting verification unit. In particular, if verification units A and B are both inherited by verification unit C, and neither A nor B inherit the other, then the names declared in A are not visible in B, and the names declared in B are not visible in A.

A verification unit A may be inherited by a verification unit B, regardless of whether the flavor of A and the flavor of B are the same or are different. If they are different, then implicit cross-language type conversions are performed as required, following the conventions for mixed-language simulation.

Inheritance only provides access to existing declarations; it does not create local copies of declarations. If a verification unit C contains a variable V, and C is inherited by two verification units A and B, then the variable C.V is visible in both A and B. Only one instance of this variable is created, and that one instance is accessible from both A and B.

Transitive closure applies by default to vunit inheritance. Thus, if vunit A inherits vunit B and vunit B inherits vunit C, then the declarations of C become visible in vunit A. Furthermore, if vunit A is bound to a design module or instance M, and vunit A is inherited in vunits B and C, then the symbols in M are visible to both B and C.

Nontransitive closure can be applied to vunit inheritance using a keyword called “nontransitive.” When nontransitive closure is specified, then only decalarations within the immediately inherited vunit are visible to the inheriting vunit. Thus if vunit A inherits vunit B using the “nontransitive” keyword, then the declarations of vunit B are visible to vunit A alone, irrespective of how vunit A is inherited by other vunits.

Inheritance Graph

An **inheritance graph** is a directed acyclic graph in which nodes are vunits and there is an edge from vunit A to vunit B if and only if A inherits B. If A inherits B using the “nontransitive” keyword, then the edge from A to B is dashed; otherwise, it is solid.

A vunit A in an inheritance graph is the **root vunit** of the sub-graph composed of all the vunits reachable from A by paths in which all edges are solid except, possibly, the last edge of the path.

Restrictions

a) If one verification unit inherits another, and both are bound to instances, then either both shall be bound to the same instance; or one shall be bound to an instance that is instantiated directly or indirectly within the instance to which the other is bound.
b) If one verification unit inherits another, and one or both are bound to a module, then the binding shall be such that if each binding to a module is considered as binding to each of its instances, the restrictions of item a) above are met.

Example

```
Module top;
       mod_b mod_b_inst();
       mod_c mod_c_inst();
endmodule

module mod_b;
       mod_d  mod_d_inst();
endmodule

module mod_c;
       mod_e mod_e_inst();
endmodule
```

Consider vunits/vmodes X and Y where X is inherited by Y.

Valid use models, assuming no other instances of module top, mod_b, and mod_c:

a) X bound to top, Y bound to either
   — instance top.mod_b_inst or
   — top.mod_c_inst or
   — to top or
   — to mod_b or
   — to mod_c

b) X bound to top, Y bound to either
   — top.mod_b_inst.mod_d_inst or
   — top.mod_c_inst.mod_e_inst

c) Y bound to top.mod_b_inst, X bound to top.mod_b_inst.mod_d_inst
d) X bound to top.mod_c_inst, Y bound to top.mod_c_inst.mod_e_inst
e) X and Y both bound to top
f) X and Y both bound to top.mod_b_inst

Invalid use model:

— X bound to top.mod_b_inst, Y bound to top.mod_c_inst

7.2.4 Overriding assignments

A verification unit may declare that it will override assignments made to a variable or signal declared elsewhere, as shown in Syntax 7-11.

```
Override_Spec  ::=
   override Name_List ;
Name_List ::=
   Name { ',' Name }
```

Syntax 7-11—Override_Spec

Any name that appears in an override specification shall be visible in the containing verification unit.

The override keyword is required. It is illegal to make an assignment to a variable or a signal declared elsewhere without specifying as such using the override keyword.

NOTE—**override** is a new keyword in the language. In IEEE Std 1850 PSL 2005 any assignment could override the behavior of a signal without having to specify it explicitly.

7.2.4.1 Simple case

In the simple case in which there is a single vunit or multiple vunits, all of which are related by hierarchy or instantiation, overriding has the following effect. If a variable or signal name N appears in the name list of an override specification, then the following effectively occurs:

a) All existing assignments to N in the design and in other verification units are redirected to assign to a new variable N'.
b) The built-in function |original(N)| returns the value of this new variable N'.
c) Any assignments to N made in this verification unit are applied to N.

As a result, assignments made to N in this verification unit effectively mask any assignments made to N elsewhere.

If an override specification is provided for a given variable or signal, and the containing verification unit does not assign to that variable or signal, then the variable or signal will act as a free variable (that is, will have a nondeterministic value).

Example

```
vunit top(mydesign) {
   override sig1;
```



```
   assign sig1 = 0;
   assert always sig2;
}
```

Assume that vunit top is the only vunit. Vunit top overrides signal `sig1` with the value 0. If `sig1` is a design signal of design mydesign, then the override effectively creates a new version of mydesign in which signal `sig1` is tied to zero (and consequently all signals in the fanout of `sig1` are influenced), and the assertion is made with respect to this version of the design.

7.2.4.2 Multiple unrelated vunits

In the case where multiple vunits are not related by hierarchy or instantiation, the override affects only those vunits that are related to the overriding vunit by hierarchy or instantiation.

Example 1

```
vunit X(mydesign) {
   override sig1;
   assign sig1 = 0;
   assert always sig2;
}
vunit Y(mydesign) {
   assert always sig3;
}
```

Vunits X and Y are not related by hierarchy or instantiation. Therefore, the override statement in vunit X affects only vunit X but does not affect vunit Y. If signal `sig1` is a design signal of design mydesign, this means that there are two versions of mydesign—one in which signal `sig1` has its original behavior (this version is seen by vunit Y) and one in which signal `sig1` is tied to 0 (this version is seen by vunit X).

If a tool does not support more than one version of a design per run, then vunits X and Y may not be run together (although they may be run separately).

Example 2

```
vunit A(mydesign) {
   inherit B;
   override sig1;
   assign sig1 = 0;
   assert always sig2;
}
vunit B(mydesign) {
   assert always sig3;
}
vunit C(mydesign) {
   inherit A;
   assert always sig4;
}
vunit D(mydesign) {
   assert always sig5;
}
```

Vunits A, B, and C are related by inheritance. Thus, if they are run together with C as the root, the assertions contained in them refer to the version of the design created by the override statement in vunit A. Vunit D is unrelated by inheritance to A, B, or C. Thus, its assertion refers to the orginal design, without the override. If a tool supports more than one version of a design per run, then vunits A, B, C, and D may be run together, otherwise vunit D shall be run separately from the others (and a tool will report an error if it is not).

Note that the version of the design referred to by vunit B depends on whether it is run as the root of an inheritance/instantiation tree (in which case its assertion refers to the original design) or together with vunit A, which inherits it (in which case it is affected by the override statement appearing in A). This means that there is no meaning to the statement "the assertions of vunit B passed." Rather, a tool shall also report the context in which a vunit was run (either as the root of an inheritance/instantiation tree or as inherited/ instantiated within another tree).

7.2.4.3 Multiple overrides to the same signal

If there are multiple overrides to the same signal, then the resolution of the multiple override statements is similar to the resolution of multiple signal declarations. That is, if the multiple overrides appear in unrelated vunits, then no resolution is necessary. However, if there are multiple overrides to the same signal in related vunits, then an override that appears in an instantiating or inheriting vunit takes precedence over an override that appears in the instantiated or inherited vunit, and if sibling vunits override the same signal but a parent vunit does not, then the result is an illegal vunit because of ambiguity.

Example 1

```
vunit top(mydesign) {
   inherit bot;
   override sig1;
   assign sig1 = 0;
   assert always sig2;
}
vunit bot(mydesign) {
   override sig1;
   assign sig1 = 1;
   assert always sig3;
}
```

Vunits top and bot both override design signal `sig1`. If vunit bot is run alone, then it uses a version of the design in which signal `sig1` is tied to 1. However, if vunit top is run as the root, then its override takes precedence, since it is the inheriting vunit, and thus it will use a version of the design in which signal `sig1` is tied to 0.

Example 2

```
vunit top(mydesign) {
   inherit A;
   inherit B;
   assert always sig2;
}
vunit A(mydesign) {
   override sig1;
   assign sig1 = 0;
   assert always sig3;
}
```



```
vunit B(mydesign) {
   override sig1;
   assign sig1 = 1;
   assert always sig4;
}
```

Vunits A and B both override signal `sig1`, and both are inherited by vunit top. Vunit top does not override signal `sig1`; therefore, the behavior of `sig1` in vunit top is ambiguous, and thus vunit top is illegal.

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 IEC/IEEE 62142. 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 8-1, 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 8-1—Integer range declaration

The nonterminals list_of_variable_identifiers and constant_expression are defined in the syntax for IEC/IEEE 62142.

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 8-2.

```
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 8-2—Structure declaration

The nonterminals list_of_variable_identifiers, net_declaration, reg_declaration, and integer_declaration are defined in the syntax for IEC/IEEE 62142.

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.

9. Scope and visibility rules

A PSL sequence declaration, property declaration, or verification unit declaration defines a name for a sequence, property, or verification unit, respectively. A PSL verification unit instantiation or PSL directive that includes a label defines the label as the name of the statement. A formal parameter list defines the names of formal parameters to be used within a parameterized construct. A forall property, a parameterized sequence, and a parameterized property each define the name of a parameter used in the replication of a sequence or property.

For each of these named objects, the defined name has a scope, i.e., a portion of the text of a PSL description in which the name can be used to refer to the corresponding object. This scope generally extends throughout the context in which the object is defined, and it may also extend beyond that context.

9.1 Immediate scope

The following rules define the *immediate* scope of each kind of name:

a) The immediate scope of the name of a verification unit is global. It extends over the set of verification units defined for use in a given context.
b) The immediate scope of the name of a sequence or property declaration extends throughout the verification unit within which it is declared.
c) The immediate scope of the label on a PSL verification unit instantiation or a PSL directive extends throughout the verification unit within which that construct occurs.
d) The immediate scope of a formal parameter extends throughout the named sequence, property, or verification unit declaration with which the formal parameter is associated.
e) The immediate scope of a replication parameter extends throughout the sequence or property within which it is defined, including any nested sub-sequences or sub-properties.

Restrictions

It is an error if a given name is defined more than once within the same immediate scope.

NOTE—As a consequence of the above rules, it is an error if there are two verification units with the same name, or two property or sequence declarations with the same name in the same verification unit, or if a given name is used as both a sequence or property name and as a label in the same verification unit, or two replication parameters with the same name defined for the same (sub) property, etc.

9.2 Extended scope

The immediate scope of a name can be extended further as a result of inheritance or instantiation. If a name is declared within a verification unit, and that verification unit is inherited by or instantiated in another verification unit, then the *extended* scope of the name also extends throughout the inheriting or instantiating verification unit.

The scope of a name declared in a design module or instance can also be extended further, as a result of binding. If the scope of a name extends to the end of a design module or instance, and a verification unit V is bound to that design module or instance, then the extended scope of that name also extends throughout the verification unit bound to that design module or instance.

9.3 Direct and indirect name references

A direct name reference is a reference that consists of just the name itself.

An indirect name reference involves a prefix that defines the scope within which the name was declared.

Within the immediate scope of a given name, a direct reference to that name is always unambiguous. This is a consequence of the definition of immediate scope and the fact that it is an error for the same name to be declared twice within a given scope.

Within the extended scope of a given name, a direct reference to that name may be illegal, because a direct reference would be ambiguous. In particular, a direct reference to a given name is illegal at a given point if the point of reference is within the immediate scope of another declaration of the same name. A direct reference to a given name would be ambiguous (and is therefore illegal) if the point of reference is within the extended scope of two or more different declarations of the same name. In either case, the given name can be referred to indirectly using a dotted name.

A dotted name consists of a prefix, followed by a '.' character, followed by a suffix. The prefix is the name of a scope: i.e., a design module name or instance pathname, a verification unit, a sequence declaration, or a property declaration. The prefix name shall be directly visible (i.e., prefix shall either be declared in immediate scope or shall represent a verification unit scope) at the point where the dotted name occurs. The suffix is a name that is defined within that scope (i.e., the scope denoted by the prefix), or is a dotted name whose prefix is a name that is defined within that scope.

Example 1

If vunit A inherits vunit B, vunit B instantiates vunit C with label C1, and each of the three vunits define sequence S, then in vunit A, one can refer to S (the one defined in A), or B.S (the one defined in B), or B.C1.S (the one defined in instance C1 of C within B).

Example 2

If vunit A defines a variable X then

a) If vunit B instantiates A, then X shall be accessed in vunit B only using an indirect reference "A.X"
b) If vunit B inherits A, then access to X in vunit B may be using a direct reference to X or using an indirect reference "A.X"

Restrictions

— A vunit name cannot appear in the suffix of a longer dotted name. It can only begin a dotted name.
— A vunit name that is the prefix of a dotted name shall be the name of a vunit that is inherited (directly or indirectly) by the enclosing vunit.
— The suffix of a given dotted name shall be visible in the vunit that is the prefix of the given name.

Examples

```
vunit A {
  sequence s = ...
  <can refer to A.s, or directly to s>
}

vunit B {
  inherit A;
  <can refer to A.s, or directly to s>
}

vunit C {
  <can refer to A or B, in order to inherit it>
  inherit B;
  <can refer to B.s  because of the transitivity of  inheritance>
  <cannot refer to B.A.s, because this is not a legal form of a dotted
name>
  <cannot refer to A.s, because A is not inherited by C>
  <can refer to s (declared in A), as there is no ambiguity on s

}
```

Annex A

(normative)

Syntax rule summary

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 | **vpkg** | **vprop** | **vmode**

e) Square brackets enclose optional items unless it appears in boldface, in which case it stands for itself. For example:

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

indicates 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.

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

i) Flavor macros, containing embedded underscores, are shown in uppercase. These reflect the various HDLs that can be used within the PSL syntax and show the definition for each HDL. The general format is the term `Flavor Macro`, then the actual *macro name*, followed by the = operator, and, finally, the definition for each of the HDLs. For example:

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

shows the *range symbol* macro (RANGE_SYM). See 4.3.2 for further details about *flavor macros*.

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

A.2 Tokens

PSL syntax is defined in terms of primitive *tokens*, which are character sequences that act as distinct symbols in the language.

Each PSL keyword is a single token. Some keywords end in one or two non-alphabetic characters (‘!’ or ‘_’ or both). Those characters are part of the keyword, not separate tokens.

Each of the following character sequences is also a token:

[**]** **(** **)** **{** **}**

, **;** **:** **..** **=** **:=**

***** **+** **|->** **|=>** **<->** **->**

[* **[+]** **[->** **[=**

&& **&** **||** **|** **!**

$ **@** **.** **/**

Finally, for a given flavor, the tokens of the corresponding HDL are tokens of PSL.

A.3 HDL dependencies

PSL depends upon the syntax and semantics of an underlying hardware description language. In particular, PSL syntax includes productions that refer to nonterminals in SystemVerilog, Verilog, VHDL, or GDL. PSL syntax also includes Flavor Macros that cause each flavor of PSL to match that of the underlying HDL for that flavor.

For SystemVerilog, the PSL syntax refers to the following nonterminals in the IEEE Std 1800 syntax:

— module_or_generate_item_declaration
— module_or_generate_item
— list_of_variable_identifiers
— identifier
— expression
— constant_expression

For Verilog, the PSL syntax refers to the following nonterminals in the IEC/IEEE 62142 syntax:

— module_or_generate_item_declaration
— module_or_generate_item
— list_of_variable_identifiers
— identifier
— expression
— constant_expression
— task_port_type

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

— block_declarative_item
— concurrent_statement
— design_unit
— identifier
— expression
— entity_aspect

For SystemC, the PSL syntax refers to the following nonterminals in the IEEE Std 1666 syntax:

— simple_type_specifier
— expression
— event_expression
— declaration
— statement
— identifier

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

— module_item_declaration
— module_item
— module_declaration
— identifier
— expression

A.3.1 Verilog extensions

For the Verilog flavor, PSL extends the forms of declaration that can be used in the modeling layer by defining two additional forms of type declaration.

Extended_Verilog_Declaration ::=
*Verilog*_module_or_generate_item_declaration
| Extended_Verilog_Type_Declaration

Extended_Verilog_Type_Declaration ::=
Finite_Integer_Type_Declaration
| Structure_Type_Declaration

Finite_Integer_Type_Declaration ::=
integer Integer_Range list_of_variable_identifiers **;**

Structure_Type_Declaration ::=
struct { Declaration_List **}** list_of_variable_identifiers **;**

Integer_Range ::=
(constant_expression **:** constant_expression **)**

Declaration_List ::=
HDL_Variable_or_Net_Declaration { HDL_Variable_or_Net_Declaration }

HDL_Variable_or_Net_Declaration ::=
net_declaration
| reg_declaration
| integer_declaration

A.3.2 Flavor macros

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

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

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: **!**

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*

Flavor Macro HDL_EXPR =
SystemVerilog: *SystemVerilog*_Expression
/ Verilog: *Verilog*_Expression
/ VHDL: *VHDL*_Expression
/ SystemC: *SystemC*_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

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

Flavor Macro HDL_DECL =
SystemVerilog: *SystemVerilog*_module_or_generate_item_declaration
/ Verilog: Extended_Verilog_Declaration
/ VHDL: *VHDL*_block_declarative_item
/ 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

Flavor Macro HDL_SEQ_ STMT =
SystemVerilog: SystemVerilog_statement_item
/ Verilog: Verilog_statement
/ VHDL: VHDL_sequential_statement
/ SystemC: SystemC_statement
/ GDL: GDL_process_item

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

Flavor Macro HDL_RANGE =
VHDL: range_attribute_name

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

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

A.4 Syntax productions

The rest of this annex defines the PSL syntax.

A.4.1 Verification units

PSL_Specification ::=
{ Verification_Item }

Verification_Unit ::=
Vunit_Type *PSL*_Identifier [**(** Context_Spec **)**] **{**
{ Inherit_Spec }
{ Override_Spec }
{ Vunit_Item }
}

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

Context_Spec ::=
Binding_Spec | Formal_Parameter_List

Binding_Spec ::=
Hierarchical_HDL_Name

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 ::=
[nontransitive] **inherit** *vunit*_Name { **,** *vunit*_Name } **;**

Vunit_Item ::=
HDL_DECL
| HDL_STMT
| PSL_Declaration
| PSL_Directive
| Vunit_Instance

Override_Spec ::=
'override' Name_List ;

Name_List ::=
Name { ',' Name }

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

A.4.2 PSL declarations

PSL_Declaration ::=
Property_Declaration
| Sequence_Declaration
| Clock_Declaration

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** | **mutable**] Value_Parameter
| **sequence**
| **property**

Value_Parameter ::=
HDL_Type
| PSL_Type_Class

HDL_Type ::=
hdltype HDL_VARIABLE_TYPE

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

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

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

Clock_Expression ::=
*boolean*_Name
| *boolean*_Built_In_Function_Call
| **(** Boolean **)**
| **(** HDL_CLOCK_EXPR **)**

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

Actual_Parameter ::=
AnyType|Number | Boolean | Property | Sequence

A.4.3 PSL directives

PSL_Directive ::=
[Label **:**] Verification_Directive
Label ::=
*PSL*_Identifier

HDL_or_PSL_Identifier ::=
*SystemVerilog*_Identifier
| *Verilog*_Identifier
| *VHDL*_Identifier
| *SystemC*_Identifier
| *GDL*_Identifier
| *PSL*_Identifier

Verification_Directive ::=
Assert_Directive
| Assume_Directive
| Restrict_Directive
| Restrict!_Directive
| Cover_Directive
| Fairness_Statement

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

Assume_Directive ::=
assume Property **;**

Restrict_Directive ::=
restrict Sequence **;**

Restrict!_Directive ::=
restrict! Sequence **;**

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

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

A.4.4 PSL properties

Property ::=
Replicator Property
| FL_Property
| OBE_Property

Replicator ::=
forall Parameter_Definition **:**

Index_Range ::=
LEFT_SYM *finite*_Range RIGHT_SYM
| **(** HDL_RANGE **)**

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

Value_Range ::=
Value
| *finite*_Range

Value ::=
Boolean
| Number

Proc_Block ::=
[[Proc_Block_Item { Proc_Block_Item }]]

Proc_Block_Item::=
HDL_DECL
| HDL_SEQ_STMT

FL_Property ::=
Boolean
| ([**[|** HDL_DECL {,HDL_DECL} **|]**] FL_Property)
| Sequence [**!**]
| *FL_property*_Name [**(** Actual_Parameter_List **)**]
| FL_Property **@** Clock_Expression
| FL_Property **abort** Boolean
| FL_Property **async_abort** Boolean
| FL_Property **sync_abort** Boolean
| Parameterized_Property
: Logical Operators :
| NOT_OP FL_Property
| FL_Property AND_OP FL_Property
| FL_Property OR_OP FL_Property
:
| FL_Property **->** FL_Property
| FL_Property **<->** FL_Property
: Primitive LTL Operators :
| **X** FL_Property
| **X!** FL_Property
| **F** FL_Property
| **G** FL_Property
| **[** FL_Property **U** FL_Property **]**
| **[** FL_Property **W** FL_Property **]**
: Simple Temporal Operators :
| **always** FL_Property
| **never** FL_Property
| **next** FL_Property
| **next!** FL_Property

| **eventually!** FL_Property
:
| FL_Property **until!** FL_Property
| FL_Property **until** FL_Property
| FL_Property **until!_** FL_Property
| FL_Property **until_** FL_Property
:
| FL_Property **before!** FL_Property
| FL_Property **before** FL_Property
| FL_Property **before!_** FL_Property
| FL_Property **before_** FL_Property

: Extended Next (Event) Operators :
| **X [** Number **] (** FL_Property **)**
| **X! [** Number **] (** FL_Property **)**
| **next [** Number **] (** FL_Property **)**
| **next! [** Number **] (** FL_Property **)**
:(see A.4.7)
| **next_a [** *finite*_Range **] (** FL_Property **)**
| **next_a! [** *finite*_Range **] (** FL_Property **)**
| **next_e [** *finite*_Range **] (** FL_Property **)**
| **next_e! [** *finite*_Range **] (** FL_Property **)**
:
| **next_event! (** Boolean **) (** FL_Property **)**
| **next_event (** Boolean **) (** FL_Property **)**
| **next_event! (** Boolean **) [** *positive*_Number **] (** FL_Property **)**
| **next_event (** Boolean **) [** *positive*_Number **] (** FL_Property **)**
:
| **next_event_a! (** Boolean **) [** *finite_positive*_Range **] (** FL_Property **)**
| **next_event_a (** Boolean **) [** *finite_positive*_Range **] (** FL_Property **)**
| **next_event_e! (** Boolean **) [** *finite_positive*_Range **] (** FL_Property **)**
| **next_event_e (** Boolean **) [** *finite_positive*_Range **] (** FL_Property **)**

: Operators on SEREs :
| { SERE } **(** FL_Property **)**
| Sequence **|->** FL_Property
| Sequence **|=>** FL_Property

A.4.5 Sequential Extended Regular Expressions (SEREs)

SERE ::=
Boolean
| Boolean Proc_Block
| Sequence
| SERE **;** SERE
| SERE **:** SERE
| Compound_SERE

Compound_SERE ::=
Repeated_SERE
| Braced_SERE
| Clocked_SERE
| Compound_SERE **|** Compound_SERE
| Compound_SERE & Compound_SERE

| Compound_SERE && Compound_SERE
| Compound_SERE **within** Compound_SERE
| Parameterized_SERE

A.4.6 Parameterized Properties and SEREs

Parameterized_Property ::=
for Parameters_Definition **:** And_Or_Property_OP **(** FL_Property **)**

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_Property_OP ::=
AND_OP
| OR_OP

And_Or_SERE_Op :: =
&& | & | |

A.4.7 Sequences

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

Repeated_SERE ::=
Boolean **[*** [Count] **]**
| Sequence **[*** [Count] **]**
| **[*** [Count] **]**
| Boolean **[+]**
| Sequence **[+]**
| **[+]**
| Boolean **[=** Count **]**
| Boolean **[->** [*positive*_Count] **]**
| Boolean Proc_Block
| Sequence Proc_Block

Braced_SERE ::=
{ [**[|** HDL DECL {HDL DECL} **|]**] SERE }
| { [free HDL Identifier {HDL Identifier}] SERE }

```
Sequence_Instance ::=
        sequence_Name [ ( Actual_Parameter_List ) ]

Clocked_SERE ::=
        Braced_SERE @ Clock_Expression

Count ::=
        Number
      | Range

Range ::=
        Low_Bound RANGE_SYM High_Bound

Low_Bound ::=
        Number
      | MIN_VAL

High_Bound ::=
        Number
      | MAX_VAL
```

A.4.8 Forms of expression

```
Any_Type ::=
        HDL_or_PSL_Expression
Bit ::=
        bit_HDL_or_PSL_Expression

Boolean ::=
        boolean_HDL_or_PSL_Expression
BitVector ::=
        bitvector_HDL_or_PSL_Expression

Number ::=
        numeric_HDL_or_PSL_Expression

String ::=
        string_HDL_or_PSL_Expression

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

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

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_Set **)**
| **nondet_vector (** Number, Value_Set **)**

Union_Expression ::=
Any_Type **union** Any_Type

A.4.9 Optional Branching Extension

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

: Logical Operators :
| NOT_OP OBE_Property
| OBE_Property AND_OP OBE_Property
| OBE_Property OR_OP OBE_Property
| OBE_Property **->** OBE_Property
| OBE_Property **<->** OBE_Property

: Universal Operators :
| **AX** OBE_Property
| **AG** OBE_Property
| **AF** OBE_Property
| **A [** OBE_Property **U** OBE_Property **]**
:
Existential Operators :
| **EX** OBE_Property
| **EG** OBE_Property
| **EF** OBE_Property
| **E [** OBE_Property **U** OBE_Property **]**

Annex B

(normative)

Formal Syntax and Semantics of IEEE Std 1850 Property Specification Language (PSL)

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

B.1 Typed-text representation of symbols

Table B.1 shows the mapping of various symbols used in this definition to the corresponding typed-text PSL representation, in the different flavors.

Table B.1: Typed-text symbols in the SystemVerilog, Verilog, VHDL, SystemC and GDL flavors

	SystemVerilog	Verilog	VHDL	SystemC	GDL
$\mapsto$	`|->`	`|->`	`|->`	`|->`	`|->`
$\Mapsto$	`|=>`	`|=>`	`|=>`	`|=>`	`|=>`
$\rightarrow$	`->`	`->`	`->`	`->`	`->`
$\leftrightarrow$	`<->`	`<->`	`<->`	`<->`	`<->`
$\neg$	`!`	`!`	`not`	`!`	`!`
$\wedge$	`&&`	`&&`	`and`	`&&`	`&`
$\vee$	`||`	`||`	`or`	`||`	`|`
..	`:`	`:`	`to`	`:`	`..`
$\langle\ \rangle$	`[ ]`	`[ ]`	`( )`	`( )`	`( )`
$\leftarrow$	`<=`	`<=`	`<=`	`=`	`:=`
$\cup$	|	|	|	|	|
$\cap$	&&	&&	&&	&&	&&
$\cdot$	;	;	;	;	;
$\circ$	:	:	:	:	:
Z$\leftarrow$E	`[[` equivalent sequence of assignments in the flavor language `]]`				
var(Z)	`[[` declaration of Z in the flavor language `]]`				
free(Z)	`[[` free(Z) `]]`				

NOTE –
For reasons of simplicity, the syntax given herein is more flexible than the one defined by the extended BNF (given in Annex A). That is, some of the expressions which are legal here are not legal under the BNF Grammar. Users should use the stricter syntax, as defined by the BNF grammar in Annex A.

B.2 Syntax

The logic PSL is defined with respect to a non-empty finite set of atomic propositions $\mathcal{P}$, a finite set of local variables $\mathcal{V}$ with a finite domain $\mathcal{D}$, a given set of (unary and binary) operators $\mathcal{O}$ over the domain $\mathcal{D}$. We assume that T and F belong to $\mathcal{D}$.

The definition of Boolean expressions, expressions and SEREs is given by mutual induction.

Definition 1 (Boolean Expressions) *Let* $\odot_{\mathrm{B}} \in \mathcal{O}$ *and* $\otimes_{\mathrm{B}} \in \mathcal{O}$ *be unary and binary operators such that* $\odot_{\mathrm{B}} : \mathcal{D} \mapsto \{\mathrm{T}, \mathrm{F}\}$ *and* $\otimes_{\mathrm{B}} : \mathcal{D}^2 \mapsto \{\mathrm{T}, \mathrm{F}\}$*. Let* $p \in \mathcal{P}$ *and let* k *be a non-negative integer. Let* e *be an expression and* r *a SERE.*

The set of Boolean expressions $\mathcal{B}$ *is defined inductively as follows:*

$$b ::= \quad p \quad | \quad \odot_{\mathrm{B}}\, e \quad | \quad e \otimes_{\mathrm{B}} e \quad | \quad prev(b, k) \quad | \quad ended(r)$$

We refer to a Boolean expression that does not use local variables and does not refer to the past (that is, does not use prev or ended) as a basic Boolean expression*.*

We use true and false as abbreviations for $p \vee \neg p$ and $p \wedge \neg p$, respectively, where p is some atomic proposition, and $\vee, \wedge \in \otimes_{\mathrm{B}}$ and $\neg \in \odot_{\mathrm{B}}$ are defined in the usual way.

Definition 2 (Expressions) *Let* $b \in \mathcal{B}$ *be a Boolean expression and* $y \in \mathcal{V}$ *be a local variable. Let* $\odot \in \mathcal{O}$ *and* $\otimes \in \mathcal{O}$ *be unary and binary operators, respectively, such that* $\odot : \mathcal{D} \mapsto \mathcal{D}$ *and* $\otimes : \mathcal{D}^2 \mapsto \mathcal{D}$*. Let* k *be a non-negative integer.*

The set of expressions $\mathcal{E}$ *is defined inductively as follows:*

$$e ::= \quad b \quad | \quad y \quad | \quad \odot\, e \quad | \quad e \otimes e \quad | \quad prev(e, k)$$

Definition 3 (Sequential Extended Regular Expressions (SEREs)) *Let* e *be an expression and* d *a value in* $\mathcal{D}$*. Let* Z *be a (possibly empty) finite sequence of local variables. Let* E *be a sequence of expressions of the same length as* Z*. Let* b *be a Boolean expression.The set of* SEREs *is defined inductively as follows:*

$$\begin{array}{rcl} r ::= & [*0] \mid e{=}d, \mathrm{Z}{\leftarrow}\mathrm{E} \mid r \cdot r \mid r \circ r \mid r[+] \mid r \cup r \mid r \cap r \\ & \{r\} \mid \{var(\mathrm{Z})\ r\} \mid \{free(\mathrm{Z})\ r\} \mid r@b \mid \end{array}$$

We sometimes use $\_\_$ to denote the empty assignment, i.e. an assignment of the form Z←E where Z is an empty sequence. For an expression e and value d, we use $e{=}d$ to abbreviate $e{=}d, \_\_$ and we use e to abbreviate $e{=}\mathrm{T}$ and $e{=}\mathrm{T}, \_\_$.

Definition 4 (Formulas) *Let* k *be a non-negative integer,* r *a SERE,* $\mathrm{Z} \subseteq \mathcal{V}$ *a finite set of local variables. Let* b *be a Boolean expression. Let* t *be a SERE that does not refer to local variables. The set of FL formulas is defined inductively as follows:*

$$\begin{array}{rcl} \varphi ::= & r! \mid r \mid r \mapsto \varphi \mid (\varphi) \mid \neg\varphi \mid \varphi \wedge \varphi \\ & X![k]\ \varphi \mid \varphi\ U\ \varphi \mid \varphi\ sync\_abort\ t \mid (var(\mathrm{Z})\ \varphi) \mid \varphi@b \end{array}$$

NOTE –
The formula φ sync_abort t is only accessible to the user when t is a Boolean expression.

Definition 5 (Formulas of the Optional Branching Extension (OBE)) *Let* b *be a basic Boolean expression. The set of OBE formulas is defined inductively as follows:*

$$f ::= \quad b \quad | \quad (f) \quad | \quad \neg f \quad | \quad f \wedge f \quad | \quad EXf \quad | \quad E[f\ U\ f] \quad | \quad EGf$$

Definition 6 (PSL Formulas)

1. *Every FL formula is a PSL formula.*
2. *Every OBE formula is a PSL formula.*

In Section B.4, we show additional operators which provide syntactic sugaring to the ones above.

B.3 Semantics

The semantics of PSL formulas are defined with respect to a *model*. A model is a quintuple $(S, S_0, R, \mathcal{P}, L)$, where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, $R \subseteq S \times S$ is the transition relation, $\mathcal{P}$ is a non-empty set of atomic propositions, and L is the valuation, a function $L : S \longrightarrow 2^{\mathcal{P}}$, mapping each state with a set of atomic propositions valid in that state.

A *path* π is a possibly empty finite (or infinite) sequence of states $\pi = (\pi_0, \pi_1, \pi_2, \cdots, \pi_n)$ (or $\pi = (\pi_0, \pi_1, \pi_2, \cdots)$). A *computation path* π of a model M is a non-empty finite (or infinite) path π such that $\pi_0 \in S_0$ and for every $i < n$, $R(\pi_i, \pi_{i+1})$ and for no s, $R(\pi_n, s)$ (or such that for every i, $R(\pi_i, \pi_{i+1})$). Given a finite (or infinite) path π, we overload L, to denote the extension of the valuation function L from states to paths as follows: $L(\pi) = L(\pi_0)L(\pi_1)\ldots L(\pi_n)$ (or $L(\pi) = L(\pi_0)L(\pi_1)\ldots$). Thus, we have a mapping from *states* in M to *letters* of $2^{\mathcal{P}}$, and from finite (or infinite) *paths* in M to finite (or infinite) *words* over $2^{\mathcal{P}}$.

B.3.1 Semantics of FL formulas

We first define the semantics without the clock operator (@), local variables, or expressions that refer to the past (*prev*() and *ended*()). Then we define the full semantics.

B.3.1.1 The Basic Semantics

The basic semantics are the semantics of formulas without the clock operator (@), local variables or expressions that refer to the past (*prev*() and *ended*()). In the basic semantics, $\mathcal{D} = \{\mathsf{true}, \mathsf{false}\}$ and all expressions are basic Boolean expressions.

We denote an element of $\mathcal{P}$ by p and an element of $\mathcal{D}$ by d. Let Σ be the set of all possible assignments to the atomic propositions $\mathcal{P}$ (i.e., $\Sigma = 2^{\mathcal{P}}$). We denote an element of Σ by σ and we use $\sigma(p)$ to denote the truth value given to p by σ.

We use u, v, w to denote (possibly empty) words over Σ. We use a, b, c to denote letters over Σ. We use ϵ to denote the empty word. We use $\cdot$ and $\circ$ to denote concatenation and concatenation with one overlapping letter, respectively. Formally, given languages L_1 and L_2 we use $L_1 \cdot L_2$ to denote the set $\{w_1 w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$. We use $L_1 \circ L_2$ to denote the set $\{w_1 \ell w_2 \mid w_1 \ell \in L_1 \text{ and } \ell w_2 \in L_2\}$. For a language L we use L^0 to denote $\{\epsilon\}$. We use L^i to denote $L^{i-1} \cdot L$. We use L^* and L^+ to denote $\bigcup_{i \geq 0} L^i$ and $\bigcup_{i > 0} L^i$, respectively. We use L^ω for the language composed of infinitely many concatenations of L with itself. We use L^∞ for the union $L^* \cup L^\omega$.

For a finite/infinite word w where $w = a_0 a_1 \cdots a_n$ or $w = a_0 a_1 a_2 \cdots$ and integers i, j we use w^i to denote the $(i+1)^{th}$ letter of w. That is, $w^i = a^i$. We use $w^{i\cdot\cdot}$ to denote the suffix of w starting at w^i. That is, $w^{i\cdot\cdot} = a_i a_{i+1} \cdots a_n$ or $w^{i\cdot\cdot} = a_i a_{i+1} \cdots$. We use $w^{i\cdot\cdot j}$ to denote the finite word starting at position i and ending at position j. That is, $w^{i\cdot\cdot j} = a_i a_{i+1} \cdots a_j$. We make use of an "overflow" and "underflow" for the positions of w. That is, $w^{i\cdot\cdot}$ and $w^{i\cdot\cdot j}$ are defined for i bigger than the last position of w and for $j < i$ as follows: $w^{i\cdot\cdot} = w^{i\cdot\cdot j} = \epsilon$.

For $d \in \{\mathrm{T}, \mathrm{F}\}$ and a (Boolean) expression e, the semantics of $e = d$, denoted $\mathcal{B}(e = d)$, is the language of all letters a on which $e = d$ holds. It is defined formally as follows.

Definition 7 (The Basic Boolean Semantics)

1. *Base case:*
   - $\mathcal{B}(p{=}d) = \{a \mid a(p) = d\}$
2. *Standard operators:*
   - $\mathcal{B}(\odot e{=}d) = \{a \mid \exists d' {\in} \mathcal{D}\ s.t.\ \odot(d'){=}d\ and\ a {\in} \mathcal{B}(e{=}d')\}$
   - $\mathcal{B}(e_1 \otimes e_2{=}d) = \{a \mid \exists d_1, d_2 \in \mathcal{D}\ s.t.\ (d_1 \otimes d_2){=}d\ and\ a {\in} \mathcal{B}(e_1{=}d_1)\ and\ a {\in} \mathcal{B}(e_2{=}d_2)\}$

For a regular expression r the basic *language* of r, denoted $\mathcal{L}(r)$, is defined as follows, where $\_\_$ denotes an empty sequence of assignments.

Definition 8 (The Basic Language of SEREs)

1. *Base cases:*
   - $\mathcal{L}([*0]) = \epsilon$
   - $\mathcal{L}(e{=}d, \_\_) = \mathcal{B}(e{=}d)$
2. *Standard SERE operators:*
   - $\mathcal{L}(\{r\}) = \mathcal{L}(r)$
   - $\mathcal{L}(r[+]) = \mathcal{L}(r)^+$
   - $\mathcal{L}(r_1{\cdot}r_2) = \mathcal{L}(r_1){\cdot}\mathcal{L}(r_2)$
   - $\mathcal{L}(r_1 \cup r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$
   - $\mathcal{L}(r_1 \circ r_2) = \mathcal{L}(r_1) \circ \mathcal{L}(r_2)$
   - $\mathcal{L}(r_1 \cap r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$

The basic *language of proper prefixes* and the basic *loop language* of r, denoted $\mathcal{F}(r)$ and $\mathcal{I}(r)$ respectively, will be used to define the semantics of a formula r (a weak SERE formula). Intuitively, $\mathcal{F}(r)$ consists of *finite* proper prefixes of words in $\mathcal{L}(r)$, except that logical contradictions such as false and structural contradictions such as $\{b\} \cap \{b{\cdot}b\}$ are considered satisfiable, and $\mathcal{I}(r)$ consists of *infinite* words in which we get "stuck forever" in a starred sub-expression of r.

Definition 9 (The Basic Language of Proper Prefixes of SEREs)

1. *Base cases:*
   - $\mathcal{F}([*0]) = \emptyset$
   - $\mathcal{F}(e = d, \_\_) = \epsilon$
2. *Standard SERE operators:*
   - $\mathcal{F}(\{r\}) = \mathcal{F}(r)$
   - $\mathcal{F}(r[+]) = \mathcal{L}(r)^*{\cdot}\mathcal{F}(r)$
   - $\mathcal{F}(r_1{\cdot}r_2) = \mathcal{F}(r_1) \cup (\mathcal{L}(r_1){\cdot}\mathcal{F}(r_2))$
   - $\mathcal{F}(r_1 \cup r_2) = \mathcal{F}(r_1) \cup \mathcal{F}(r_2)$
   - $\mathcal{F}(r_1 \circ r_2) = \mathcal{F}(r_1) \cup (\mathcal{L}(r_1) \circ \mathcal{F}(r_2))$
   - $\mathcal{F}(r_1 \cap r_2) = \mathcal{F}(r_1) \cap \mathcal{F}(r_2)$

Definition 10 (The Basic Loop Language of SEREs)

1. *Base cases:*
   - $\mathcal{I}([*0]) = \emptyset$
   - $\mathcal{I}(e{=}d, \_\_) = \emptyset$
2. *Standard SERE operators:*
   - $\mathcal{I}(\{r\}) = \mathcal{I}(r)$
   - $\mathcal{I}(r[+]) = (\mathcal{L}(r)^*{\cdot}\mathcal{I}(r)) \cup \mathcal{L}(r)^\omega$
   - $\mathcal{I}(r_1{\cdot}r_2) = \mathcal{I}(r_1) \cup (\mathcal{L}(r_1){\cdot}\mathcal{I}(r_2))$
   - $\mathcal{I}(r_1 \cup r_2) = \mathcal{I}(r_1) \cup \mathcal{I}(r_2)$
   - $\mathcal{I}(r_1 \circ r_2) = \mathcal{I}(r_1) \cup (\mathcal{L}(r_1) \circ \mathcal{I}(r_2))$
   - $\mathcal{I}(r_1 \cap r_2) = \mathcal{I}(r_1) \cap \mathcal{I}(r_2)$

We define the semantics of a formula with respect to a word w and a context indicating the view $\upsilon \in \{\text{S}, \text{N}, \text{W}\}$ (where S, N, W correspond to the *strong*, *neutral* and *weak* views, respectively). For a view υ we use $\overline{\upsilon}$ to denote the *dual view* which is S if $\upsilon = \text{W}$, W if $\upsilon = \text{S}$, and υ otherwise (if $\upsilon = \text{N}$). We use j and k below to denote non-negative integers.

The base cases for the definitions of a formula are strong and weak SEREs: $r!$ and r. A strong SERE $r!$ holds under the neutral or strong view if it has a non-empty prefix in the language of r. A strong SERE $r!$ holds under the weak view if it holds under the neutral view or the word "'ended too soon" — that is, the word is a proper prefix of a word in the language of r (formally, $w \in \mathcal{F}(r) \cup \{\epsilon\}$). A weak SERE r holds on a given word if the strong SERE r holds on that word or the word "got stuck forever" in a starred sub-SERE of r. That is, $w \in \mathcal{I}(r)$. In addition, under the weak and neutral views the word may "end too soon".

The relation $w \models^{\upsilon} \varphi$, read "$w$ models φ in the view υ under the basic semantics", is formally defined as follows.

Definition 11 (Satisfaction in the Basic Semantics)

I. $w \models^{\upsilon} r! \iff$ *either* $\exists j$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$
or ($\upsilon = \text{W}$ *and* $w \in \mathcal{F}(r) \cup \{\epsilon\}$)

II. $w \models^{\upsilon} r \iff$ *either* $\exists j$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$ *or* $w \in \mathcal{I}(r)$
or ($\upsilon \in \{\text{W}, \text{N}\}$ *and* $w \in \mathcal{F}(r) \cup \{\epsilon\}$)

III. $w \models^{\upsilon} r \mapsto \varphi \Longleftrightarrow \forall j$ *s.t.* $w^{0..j} \in \mathcal{L}(r)$ *we have that* $w^{j..} \models^{\upsilon} \varphi$ *and either* $\upsilon \in \{\mathrm{W}, \mathrm{N}\}$ *or* $w \notin \mathcal{F}(r) \cup \{\epsilon\}$

IV. $w \models^{\upsilon} (\varphi) \Longleftrightarrow w \models^{\upsilon} \varphi$

V. $w \models^{\upsilon} \neg\varphi \Longleftrightarrow w \not\models^{\overline{\upsilon}} \varphi$

VI. $w \models^{\upsilon} \varphi \wedge \psi \Longleftrightarrow w \models^{\upsilon} \varphi$ *and* $w \models^{\upsilon} \psi$

VII. $w \models^{\upsilon} \mathsf{X!}[k]\,\varphi \Longleftrightarrow$ *either* $w^{0..k} \in \mathcal{L}(true\,[k{+}1])$ *and* $w^{k..} \models^{\upsilon} \varphi$ *or* $(\upsilon = \mathrm{W}$ *and* $w \in \mathcal{F}(true\,[k{+}1]) \cup \{\epsilon\})$

VIII. $w \models^{\upsilon} \varphi\,\mathsf{U}\psi \Longleftrightarrow \exists k$ *s.t.* $w \models^{\upsilon} \mathsf{X!}[k]\,\psi$ *and* $\forall j < k,\ w \models^{\upsilon} \mathsf{X!}[j]\,\varphi$

IX. $w \models^{\upsilon} \varphi$ *sync_abort* $t \Longleftrightarrow$ *either* $w \models^{\upsilon} \varphi$ *or* $\exists j, k$ *s.t.* $w^{j..k} \in \mathcal{L}(t)$ *and* $w^{0..k-1} \models^{\mathrm{W}} \varphi$

Definition 12 (Holds Weakly, Neutrally, Strongly, Pending and Fails in the Basic Semantics) *Let w be a word and φ a PSL formula.*

- φ holds weakly *on w in the basic semantics if* $w \models^{\mathrm{W}} \varphi$.
- φ holds (neutrally) *on w in the basic semantics if* $w \models^{\mathrm{N}} \varphi$.
- φ holds strongly *on w in the basic semantics if* $w \models^{\mathrm{S}} \varphi$.
- φ *is* pending *on word w iff* $w \models^{\mathrm{W}} \varphi$ *and* $w \not\models^{\mathrm{N}} \varphi$
- φ fails *on word w iff* $w \not\models^{\mathrm{W}} \varphi$

NOTES –

I. The semantics given here for the LTL operators under the neutral view and in the absence of non-degenerate SEREs is equivalent to the standard LTL semantics Manna and Pnueli [B10], Pnueli [B11]. In particular, the semantics of X! and U under the neutral view can be equivalently phrased as follows:
   - $w \models^{\mathrm{N}} \mathsf{X!}\,[k]\,\varphi \Longleftrightarrow |w| > k$ and $w^{k..} \models^{\mathrm{N}} \varphi$
   - $w \models^{\mathrm{N}} \varphi\,\mathsf{U}\,\psi \Longleftrightarrow \exists k$ s.t. $w^{k..} \models^{\mathrm{N}} \psi$ and $\forall j < k,\ w^{j..} \models^{\mathrm{N}} \varphi$

   The semantics of these operators are phrased as they are in Definition 11 since this form is more similar to the one for full semantics.

II. The semantics given here for the LTL operators and the sync_abort operator is equivalent to the truncated semantics given in Eisner et al. [B1] where $\models^{\mathrm{S}}$, $\models^{\mathrm{N}}$ and $\models^{\mathrm{W}}$ are denoted $\models^{+}$, $\models$ and $\models^{-}$, respectively, and sync_abort is denoted trunc_w.

III. For FL formulas without the $\cap$ operator, the semantics here is equivalent to the semantics given in the previous version of the standard (IEEE 1850-2005) where words are interpreted over $2^P \cup \{\top, \bot\}$. Then, as shown in Eisner et al. [B3], the three following equivalences hold for a formula φ of LTL^{trunc}:

$$w \models^{\mathrm{W}} \varphi \Longleftrightarrow w\top^{\omega} \models \varphi$$
$$w \models^{\mathrm{N}} \varphi \Longleftrightarrow w \models \varphi$$
$$w \models^{\mathrm{S}} \varphi \Longleftrightarrow w\bot^{\omega} \models \varphi$$

IV. As in Eisner et al. [B1], for an infinite word w and a formula φ the three views coincide. That is, $w \models^{\mathrm{W}} \varphi \Longleftrightarrow w \models^{\mathrm{N}} \varphi \Longleftrightarrow w \models^{\mathrm{S}} \varphi$ when w is infinite.

V. There is a subtle difference between Boolean negation and formula negation. For instance, consider the formula $\neg b$. If $\neg$ is Boolean negation, then $\neg b$ is equivalent to formula $\{\neg b\}$, which holds on an empty path. If $\neg$ is formula negation, then $\neg b$ is equivalent to formula $\neg\{b\}$, which does not hold on an empty path. Rather than introduce distinct operators for Boolean and formula negation, we instead adopt the convention that negation applied to a Boolean expression is Boolean negation. This does not restrict expressivity, as braces can be used to get formula negation.

VI. For FL formulas including the $\cap$ operator, the semantics here differs from the semantics given in the previous version of the standard (IEEE 1850-2005) for formulas including *structural contradictions* such as $\{a\} \cap \{a{\cdot}a\}$. In the previous version of the standard, the formula $\{a{\cdot}b[*]{\cdot}r\}$ holds weakly on an infinite path such that a holds on the first letter and b on all the rest if r is a logical contradiction such as false, but not if r is a structural contradiction such as $\{c\} \cap \{c{\cdot}c\}$. In the semantics presented here, logical and structural contradictions are treated in a consistent manner. We achieve this by eliminating letters $\top$ and $\bot$ and including instead a context which represents the view (strong, neutral or weak) and three languages for a SERE. The language $\mathcal{L}(r)$ corresponds to tight satisfaction of the previous version of the semantics. Intuitively, the *language of proper prefixes* $\mathcal{F}(r)$ consists of finite proper prefixes of words in $\mathcal{L}(r)$ except that logical and structural contradictions are considered satisfiable. The *loop language* $\mathcal{I}(r)$ consists of infinite words in which we get "stuck forever" in a starred subexpression of r. See Eisner and Fisman [B6] for a thorough study of this subject.

B.3.1.2 The Full Semantics

In the full semantics we add the following to the basic semantics: the clock operator (@), local variables, and expressions that refer to the past (*prev*() and *ended*()).

B.3.1.2.1 Supporting the clock operator @

In order to support the clock operator @, we add a context κ — a Boolean expression that remembers the current clock. In addition, we define three languages *no_tick*, *tick* and *past_tick*, each of which uses the clock context (see Definition 19).

NOTE –
The clocked semantics for the LTL subset follows Eisner et al. [B2], with the exception that strength is applied at the SERE level rather than at the propositional level. Furthermore, following Fisman [B8] clock alignment operators are added. The clock strong and weak alignment operators are X![0] and X[0], respectively.

B.3.1.2.2 Supporting local variables

In order to support local variables we use an enhanced alphabet Υ as defined below, and we add a context Y that remembers the set of local variables currently in scope.

We denote an element of $\mathcal{P}$ by p, an element of $\mathcal{V}$ by x, y or z, and an element of $\mathcal{D}$ by d. Recall that $\mathcal{D}$ can be any finite superset of $\{\mathrm{T}, \mathrm{F}\}$. Let Σ be the set of all possible assignments to the atomic propositions $\mathcal{P}$ (i.e., $\Sigma = 2^{\mathcal{P}}$). Let Γ be the set of all possible valuations of the local variables in $\mathcal{V}$ (i.e., $\Gamma = \mathcal{D}^{\mathcal{V}}$). We denote an element of Σ by σ and an element of Γ by γ. We use $\sigma(p)$ to denote the truth value given to p by σ, and $\gamma(y)$ to denote the value given to y by γ.

We use Υ to denote the alphabet $\Sigma \times \Gamma \times \Gamma$. We use $\mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{h}$ to denote (possibly empty) words over Υ. We use $\mathtt{a}, \mathtt{b}, \mathtt{c}$ to denote letters over Υ. We refer to letters and words over Υ as *enhanced letters* and *enhanced words*, respectively. Let $\mathtt{w} = (\sigma_0, \gamma_0, \gamma'_0)(\sigma_1, \gamma_1, \gamma'_1)\cdots$ be an enhanced word. We use $\mathtt{w}|_\sigma$,$\mathtt{w}|_\gamma$,$\mathtt{w}|_{\gamma'}$ to denote the word obtained from $\mathtt{w}$ by leaving only the first, second or third component, respectively. That is $\mathtt{w}|_\sigma = \sigma_0\sigma_1\cdots$, $\mathtt{w}|_\gamma = \gamma_0\gamma_1\cdots$ and $\mathtt{w}|_{\gamma'} = \gamma'_0\gamma'_1\cdots$.

The intuitive roles of the components of a letter are as follows. The first component of a letter $\mathtt{a}$ (i.e. $\mathtt{a}|_\sigma$) holds the valuation of the atomic propositions at the given cycle. The second component of a letter $\mathtt{a}$ (i.e. $\mathtt{a}|_\gamma$) holds the current values of the local variables at the given cycle (the *pre-value*). The third component of a letter $\mathtt{a}$ (i.e. $\mathtt{a}|_{\gamma'}$) holds the values of the local variables after the assignments have taken place (the *post-value*).

NOTE –
The semantics for local variables follows the semantics proposed in Eisner and Fisman [B5,B7].

B.3.1.2.3 Supporting past expressions *prev*() and *ended*()

In order to support past expressions *prev*() and *ended*() we use *words with history* defined as follows.

We refer to a pair of words $(\mathtt{h}, \mathtt{w})$ as a word with history (or *h-word* for short) where intuitively $\mathtt{h}$ is the history and $\mathtt{w}$ is the present and future, with the first letter of $\mathtt{w}$ being the present. We refer to a pair $(\mathtt{h}, \mathtt{a})$ as an *h-letter*. We use $\mathfrak{a}$ and $\mathfrak{B}$ to denote h-words.

For standard words w_1 and w_2 from some alphabet, concatenation means simply placing the letters of w_2 after the letters of w_1 to get the word $w = w_1 w_2$. Concatenating h-words is similar, but we must account for history. Intuitively, the historical component $\mathtt{h}$ of an h-word $(\mathtt{h}, \mathtt{w})$ records the historical context of $\mathtt{w}$, for use in case we encounter a *prev* or *ended* operator. For concatenation to be consistent, then, concatenating h-words $(\mathtt{h}_1, \mathtt{w}_1)$ and $(\mathtt{h}_2, \mathtt{w}_2)$ should only be possible if $\mathtt{h}_2$ correctly records the history seen by the resulting word just before it encounters $\mathtt{w}_2$. That is, if $\mathtt{h}_2 = \mathtt{h}_1\mathtt{w}_1$. Fusion of h-words is similar to concatenation, except that there is a one-letter overlap between the words being concatenated, where intuitively, the overlap letter is actually the merging of two separate letters such that the σ components of the two letters agree, but the γ and γ' components do not necessarily agree. This allows us to support an assignment taking place "in the middle of a letter", as we desire.

Definition 13 (Concatenation and fusion of languages of h-words) *Let L_1 and L_2 be sets of h-words. Their concatenation and fusion is defined as follows, where $\cdot$ and $\circ$ denote concatenation and fusion, respectively.*
- $L_1 \cdot L_2 = \{(\mathtt{h}_1, \mathtt{u}_1\mathtt{u}_2) \mid (\mathtt{h}_1, \mathtt{u}_1) \in L_1 \text{ and } (\mathtt{h}_1\mathtt{u}_1, \mathtt{u}_2) \in L_2\}$
- $L_1 \circ L_2 = \{(\mathtt{h}_1, \mathtt{v}_1\mathtt{a}\mathtt{v}_2) \mid \exists\tilde{\gamma} \text{ such that } (\mathtt{h}_1, \mathtt{v}_1(\mathtt{a}|_\sigma, \mathtt{a}|_\gamma, \tilde{\gamma})) \in L_1 \text{ and } (\mathtt{h}_1\mathtt{v}_1, (\mathtt{a}|_\sigma, \tilde{\gamma}, \mathtt{a}|_{\gamma'})\mathtt{v}_2) \in L_2\}$

We use $\mathbb{\lambda}$ to denote the set $\{(\mathtt{h}, \epsilon) \mid \mathtt{h} \in \Upsilon^*\}$ where ϵ denotes the empty word. For a language L of h-words we use L^0 to denote the set $\mathbb{\lambda}$. We use L^i to denote $L^{i-1} \cdot L$. We use L^* and L^+ to denote $\bigcup_{i\geq 0} L^i$ and $\bigcup_{i>0} L^i$, respectively. We use L^ω for the language composed of infinitely many concatenations of L with itself. We use L^∞ for the union $L^* \cup L^\omega$.

For a finite/infinite h-word $\mathfrak{a} = (\mathtt{h}, \mathtt{w})$ where $\mathtt{w} = \mathtt{a}_0\mathtt{a}_1 \cdots \mathtt{a}_n$ or $\mathtt{w} = \mathtt{a}_0\mathtt{a}_1\mathtt{a}_2 \cdots$ and integers j, k we use $\mathfrak{a}^i$ to denote the letter $(i{+}1)^{th}$ letter of $\mathtt{w}$. That is, $\mathfrak{a}^i = \mathtt{a}^i$. We use $\mathfrak{a}^{i\cdot\cdot}$ to denote the suffix of $\mathfrak{a}$ starting at $\mathfrak{a}^i$. That is, $\mathfrak{a}^{i\cdot\cdot} = (\mathtt{h}\mathtt{a}_0\mathtt{a}_1 \cdots \mathtt{a}_{i-1},\ \mathtt{a}_i\mathtt{a}_{i+1} \cdots \mathtt{a}_n)$ or $\mathfrak{a}^{i\cdot\cdot} = (\mathtt{h}\mathtt{a}_0\mathtt{a}_1 \cdots \mathtt{a}_{i-1},\ \mathtt{a}_i\mathtt{a}_{i+1} \cdots)$. We use $\mathfrak{a}^{i..j}$ to denote the finite word starting at position i and ending at position j. That is, $\mathfrak{a}^{i..j} = (\mathtt{h}\mathtt{a}_0\mathtt{a}_1 \cdots \mathtt{a}_{i-1},\ \mathtt{a}_i\mathtt{a}_{i+1} \cdots \mathtt{a}_j)$. We make use of an "overflow" and "underflow" for the positions of $\mathfrak{a}$. That is, $\mathfrak{a}^{j\cdot\cdot}$ and $\mathfrak{a}^{j..k}$ are defined for j bigger than the last position of $\mathfrak{a}$ and for $k < j$ as follows: $\mathfrak{a}^{j\cdot\cdot} = \mathfrak{a}^{j..k} = (\mathtt{h}\mathtt{w}, \epsilon)$.

B.3.1.2.4 The Full Semantics

For a (Boolean) expression $e = d$, the Boolean semantics of $e = d$ with respect to a given set of local variables Y and a clock context κ, denoted $\mathcal{B}_{\kappa,\mathrm{Y}}(e)$, is the language of all h-letters $(\mathtt{h}, \mathtt{a})$ on which $e = d$ holds. It is defined formally as follows. The definition is given by mutual induction with the language of SEREs, denoted $\mathcal{L}_{\kappa,\mathrm{Y}}(\cdot)$, given in Definition 20 and the definition of ticks given in Definition 19.

Definition 14 (The Boolean Semantics)

1. *Base cases:*
   - $\mathcal{B}_{\kappa,\mathrm{Y}}(p{=}d) = \{(\mathtt{h}, \mathtt{a}) \mid \mathtt{a}|_\sigma(p) = d\}$
   - $\mathcal{B}_{\kappa,\mathrm{Y}}(y{=}d) = \{(\mathtt{h}, \mathtt{a}) \mid \mathtt{a}|_\gamma(y) = d\}$
2. *Standard operators:*
   - $\mathcal{B}_{\kappa,\mathrm{Y}}(\odot e{=}d) = \{(\mathtt{h}, \mathtt{a}) \mid \exists d' \in \mathcal{D} \text{ s.t. } \odot(d'){=}d \text{ and } (\mathtt{h}, \mathtt{a}){\in}\mathcal{B}_{\kappa,\mathrm{Y}}(e{=}d')\}$
   - $\mathcal{B}_{\kappa,\mathrm{Y}}(e_1 \otimes e_2{=}d) = \left\{(\mathtt{h}, \mathtt{a}) \;\middle|\; \begin{array}{l} \exists d_1, d_2 \in \mathcal{D} \text{ s.t. } (d_1 \otimes d_2){=}d \text{ and} \\ (\mathtt{h}, \mathtt{a}){\in}\mathcal{B}_{\kappa,\mathrm{Y}}(e_1{=}d_1) \text{ and } (\mathtt{h}, \mathtt{a}){\in}\mathcal{B}_{\kappa,\mathrm{Y}}(e_2{=}d_2) \end{array}\right\}$
3. *Past expressions:*
   - $\mathcal{B}_{\kappa,\mathrm{Y}}(prev(e, n){=}d) = \left\{(\mathtt{h}, \mathtt{a}) \;\middle|\; \begin{array}{l} \exists \mathtt{u}, \mathtt{v} \in \Upsilon^*\ \exists \mathtt{b} \in \Upsilon \text{ s.t. } \mathtt{h}\mathtt{a} = \mathtt{u}\mathtt{b}\mathtt{v} \text{ and} \\ (\mathtt{u}, \mathtt{b}\mathtt{v}) \in past\_tick_{\kappa,\mathrm{Y}}^{n+1} \text{ and } (\mathtt{u}, \mathtt{b}) \in \mathcal{B}_{\kappa,\mathrm{Y}}(e{=}d) \end{array}\right\}$

– $\mathcal{B}_{\kappa,\mathrm{Y}}(ended(r)\text{=}\mathrm{T}) = \{(\mathbb{h},\mathbb{a}) \mid \exists \mathbb{u},\mathbb{v} \in \Upsilon^* \text{ s.t. } \mathbb{h} = \mathbb{u}\mathbb{v} \text{ and } (\mathbb{u},\mathbb{v}\mathbb{a}) \in \mathcal{L}_{\kappa,\mathrm{Y}}(r)\}$

Definition 15 (The value of an expression) *For an h-letter* $(\mathbb{h},\mathbb{a})$ *and an expression* e *we use* $e_{\kappa,\mathrm{Y}}(\mathbb{h},\mathbb{a})$ *to denote the single element of the language* $\{d \mid (\mathbb{h},\mathbb{a}) \in \mathcal{B}_{\kappa,\mathrm{Y}}(e\text{=}d)\}$.

Definition 16 (Agrees Relative to Y) *Let* $\gamma_1,\gamma_2 \in \Gamma$ *and* $\mathrm{Y} \subseteq \mathcal{V}$. *We say that* γ_1 agrees *with* γ_2 *relative to* Y, *denoted* $\gamma_1 \overset{\mathrm{Y}}{\sim} \gamma_2$, *if for every* $y \in \mathrm{Y}$ *we have* $\gamma_1(y) = \gamma_2(y)$.

Given a sequence of assignments Z←E where $\mathrm{Z} = \{z_1, z_2, \ldots, z_n\}$ is a finite (possibly empty) set of local variables and $\mathrm{E} = \{e_1, e_2 \ldots, e_n\}$ is a sequence of assignments of the same length, we would like to compute the results of all these assignments taking place one after the other. That is, first the assignment $z_1 \leftarrow e_1$ takes place, then the assignment $z_2 \leftarrow e_2$ takes place etc., until the assignment $z_n \leftarrow e_n$ takes place.

Definition 17 (Sequence of Assignments) *Let* κ *be a Boolean expression and* $\mathrm{Y} \subseteq \mathcal{V}$. *Let* z *be a local variable and* e *an expression. Let* $\mathrm{Z} = z_1, \ldots, z_n$ *be a sequence of local variables and* E *a sequence of expressions* $\mathrm{E} = e_1, \ldots, e_n$ *of the same length.*

– *We write* $[z \leftarrow e]_{\kappa,\mathrm{Y}}(\mathbb{h},\mathbb{a})$ *to denote the valuation* $\hat{\gamma}$ *such that* $\hat{\gamma}(z) = e_{\kappa,\mathrm{Y}}(\mathbb{h},\mathbb{a})$ *and for every local variable* $y \in \mathcal{V}\backslash\{z\}$ *we have that* $\hat{\gamma}(y) = \gamma(y)$.

– *We write* $[z_1 \leftarrow e_1, \ldots, z_n \leftarrow e_n]_{\kappa,\mathrm{Y}}(\mathbb{h},\mathbb{a})$ *to denote the recursive application*

$$[z_2 \leftarrow e_2, \ldots, z_n \leftarrow e_n]_{\kappa,\mathrm{Y}}(\mathbb{h}, \langle \mathbb{a}|_\sigma, \hat{\gamma}, \mathbb{a}|_{\gamma'} \rangle)$$

where $\hat{\gamma} = [z_1 \leftarrow e_1]_{\kappa,\mathrm{Y}}(\mathbb{h},\mathbb{a})$.

We write Z←E *to abbreviate* $z_1 \leftarrow e_1, \ldots, z_n \leftarrow e_n$. *If* Z *is empty we read* Z←E *as the empty assignment* $\_\_$ *and understand* $[\mathrm{Z} \leftarrow \mathrm{E}]_{\kappa,\mathrm{Y}}(\mathbb{h},\mathbb{a})$ *to be* $\mathbb{a}|_\gamma$.

The *assignment language* of Z←E formally defined below returns the set of all h-letters $(\mathbb{h},\mathbb{a})$ such that $\mathbb{a}$ records correctly the assignment Z←E taking place at on $\mathbb{a}$ given the past is $\mathbb{h}$.

Definition 18 (The Assignment Language) *Let* κ *be a Boolean expression and* $\mathrm{Y} \subseteq \mathcal{V}$. *Let* Z *be a sequence of local variables and* E *a sequence of expressions* E *of the same length. The assignment language of* Z←E *with respect to* κ, Y *is defined as follows.*

– $\mathcal{A}_{\kappa,\mathrm{Y}}(\mathrm{Z} \leftarrow \mathrm{E}) = \{(\mathbb{h},\mathbb{a}) \mid \mathbb{a}|_{\gamma'} \overset{\mathrm{Y}}{\sim} [\mathrm{Z} \leftarrow \mathrm{E}]_{\kappa,\mathrm{Y}}(\mathbb{h},\mathbb{a})\}$

Definition 19 (no_tick,tick,past_tick) *Let* κ *be a Boolean expression and* $\mathrm{Y} \subseteq \mathcal{V}$. *We define the languages* $no\_tick_{\kappa,\mathrm{Y}}$, $tick_{\kappa,\mathrm{Y}}$ *and* $past\_tick_{\kappa,\mathrm{Y}}$ *as follows:*

– $no\_tick_{\kappa,\mathrm{Y}} = (\mathcal{B}_{\mathit{true},\mathrm{Y}}(\neg\kappa) \cap \mathcal{A}_{\kappa,\mathrm{Y}}(\_\_))$

– $tick_{\kappa,\mathrm{Y}} = (no\_tick_{\kappa,\mathrm{Y}})^* \cdot \mathcal{B}_{\mathit{true},\mathrm{Y}}(\kappa)$

– $past\_tick_{\kappa,\mathrm{Y}} = \mathcal{B}_{\mathit{true},\mathrm{Y}}(\kappa) \cdot (no\_tick_{\kappa,\mathrm{Y}})^*$

For a regular expression r the *language* of r with respect to κ, Y, denoted $\mathcal{L}_{\kappa,\mathrm{Y}}(r)$, is defined by mutual induction with Definition 14 as follows.

Definition 20 (The Language of SEREs)

1. *Base cases:*
   – $\mathcal{L}_{\kappa,\mathrm{Y}}([*0]) = \mathbb{\lambda}$
   – $\mathcal{L}_{\kappa,\mathrm{Y}}(e\text{=}d, \mathrm{Z} \leftarrow \mathrm{E}) = tick_{\kappa,\mathrm{Y}} \cap (\Upsilon^* \cdot (\mathcal{B}_{\kappa,\mathrm{Y}}(e\text{=}d) \cap \mathcal{A}_{\kappa,\mathrm{Y}}(\mathrm{Z} \leftarrow \mathrm{E})))$

2. *Standard SERE operators:*
   - $\mathcal{L}_{\kappa,\mathrm{Y}}(\{r\}) = \mathcal{L}_{\kappa,\mathrm{Y}}(r)$
   - $\mathcal{L}_{\kappa,\mathrm{Y}}(r_1{\cdot}r_2) = \mathcal{L}_{\kappa,\mathrm{Y}}(r_1){\cdot}\mathcal{L}_{\kappa,\mathrm{Y}}(r_2)$
   - $\mathcal{L}_{\kappa,\mathrm{Y}}(r_1{\circ}r_2) = \mathcal{L}_{\kappa,\mathrm{Y}}(r_1){\circ}\mathcal{L}_{\kappa,\mathrm{Y}}(r_2)$
   - $\mathcal{L}_{\kappa,\mathrm{Y}}(r[+]) = \mathcal{L}_{\kappa,\mathrm{Y}}(r)^{+}$
   - $\mathcal{L}_{\kappa,\mathrm{Y}}(r_1 \cup r_2) = \mathcal{L}_{\kappa,\mathrm{Y}}(r_1) \cup \mathcal{L}_{\kappa,\mathrm{Y}}(r_2)$
   - $\mathcal{L}_{\kappa,\mathrm{Y}}(r_1 \cap r_2) = \mathcal{L}_{\kappa,\mathrm{Y}}(r_1) \cap \mathcal{L}_{\kappa,\mathrm{Y}}(r_2)$

3. *Context operators:*
   - $\mathcal{L}_{\kappa,\mathrm{Y}}(\{\mathsf{var}(\mathrm{Z})\ r\}) = \mathcal{L}_{\kappa,\mathrm{Y}\cup\mathrm{Z}}(r)$
   - $\mathcal{L}_{\kappa,\mathrm{Y}}(\{\mathsf{free}(\mathrm{Z})\ r\}) = \mathcal{L}_{\kappa,\mathrm{Y}\setminus\mathrm{Z}}(r)$
   - $\mathcal{L}_{\kappa,\mathrm{Y}}(r\,@b) = \mathcal{L}_{b,\mathrm{Y}}(r)$

Definition 21 (The Language of Proper Prefixes of SEREs)

1. *Base cases:*
   - $\mathcal{F}_{\kappa,\mathrm{Y}}([*0]) = \emptyset$
   - $\mathcal{F}_{\kappa,\mathrm{Y}}(e = d, \mathrm{Z}{\leftarrow}\mathrm{E}) = (no\_tick_{\kappa,\mathrm{Y}})^{*}$

2. *Standard SERE operators:*
   - $\mathcal{F}_{\kappa,\mathrm{Y}}(\{r\}) = \mathcal{F}_{\kappa,\mathrm{Y}}(r)$
   - $\mathcal{F}_{\kappa,\mathrm{Y}}(r_1{\cdot}r_2) = \mathcal{F}_{\kappa,\mathrm{Y}}(r_1) \cup (\mathcal{L}_{\kappa,\mathrm{Y}}(r_1){\cdot}\mathcal{F}_{\kappa,\mathrm{Y}}(r_2))$
   - $\mathcal{F}_{\kappa,\mathrm{Y}}(r_1{\circ}r_2) = \mathcal{F}_{\kappa,\mathrm{Y}}(r_1) \cup (\mathcal{L}_{\kappa,\mathrm{Y}}(r_1){\circ}\mathcal{F}_{\kappa,\mathrm{Y}}(r_2))$
   - $\mathcal{F}_{\kappa,\mathrm{Y}}(r[+]) = \mathcal{L}_{\kappa,\mathrm{Y}}(r)^{*}{\cdot}\mathcal{F}_{\kappa,\mathrm{Y}}(r)$
   - $\mathcal{F}_{\kappa,\mathrm{Y}}(r_1 \cup r_2) = \mathcal{F}_{\kappa,\mathrm{Y}}(r_1) \cup \mathcal{F}_{\kappa,\mathrm{Y}}(r_2)$
   - $\mathcal{F}_{\kappa,\mathrm{Y}}(r_1 \cap r_2) = \mathcal{F}_{\kappa,\mathrm{Y}}(r_1) \cap \mathcal{F}_{\kappa,\mathrm{Y}}(r_2)$

3. *Context operators:*
   - $\mathcal{F}_{\kappa,\mathrm{Y}}(\{\mathsf{var}(\mathrm{Z})\ r\}) = \mathcal{F}_{\kappa,\mathrm{Y}\cup\mathrm{Z}}(r)$
   - $\mathcal{F}_{\kappa,\mathrm{Y}}(\{\mathsf{free}(\mathrm{Z})\ r\}) = \mathcal{F}_{\kappa,\mathrm{Y}\setminus\mathrm{Z}}(r)$
   - $\mathcal{F}_{\kappa,\mathrm{Y}}(r\,@b) = \mathcal{F}_{b,\mathrm{Y}}(r)$

Definition 22 (The Loop Language of SEREs)

1. *Base cases:*
   - $\mathcal{I}_{\kappa,\mathrm{Y}}([*0]) = \emptyset$
   - $\mathcal{I}_{\kappa,\mathrm{Y}}(e{=}d, \mathrm{Z}{\leftarrow}\mathrm{E}) = (no\_tick_{\kappa,\mathrm{Y}})^{\omega}$

2. *Standard SERE operators:*
   - $\mathcal{I}_{\kappa,\mathrm{Y}}(\{r\}) = \mathcal{I}_{\kappa,\mathrm{Y}}(r)$
   - $\mathcal{I}_{\kappa,\mathrm{Y}}(r_1{\cdot}r_2) = \mathcal{I}_{\kappa,\mathrm{Y}}(r_1) \cup (\mathcal{L}_{\kappa,\mathrm{Y}}(r_1){\cdot}\mathcal{I}_{\kappa,\mathrm{Y}}(r_2))$
   - $\mathcal{I}_{\kappa,\mathrm{Y}}(r_1{\circ}r_2) = \mathcal{I}_{\kappa,\mathrm{Y}}(r_1) \cup (\mathcal{L}_{\kappa,\mathrm{Y}}(r_1){\circ}\mathcal{I}_{\kappa,\mathrm{Y}}(r_2))$
   - $\mathcal{I}_{\kappa,\mathrm{Y}}(r[+]) = (\mathcal{L}_{\kappa,\mathrm{Y}}(r)^{*}{\cdot}\mathcal{I}_{\kappa,\mathrm{Y}}(r)) \cup \mathcal{L}_{\kappa,\mathrm{Y}}(r)^{\omega}$
   - $\mathcal{I}_{\kappa,\mathrm{Y}}(r_1 \cup r_2) = \mathcal{I}_{\kappa,\mathrm{Y}}(r_1) \cup \mathcal{I}_{\kappa,\mathrm{Y}}(r_2)$
   - $\mathcal{I}_{\kappa,\mathrm{Y}}(r_1 \cap r_2) = \mathcal{I}_{\kappa,\mathrm{Y}}(r_1) \cap \mathcal{I}_{\kappa,\mathrm{Y}}(r_2)$

3. *Context operators:*
   - $\mathcal{I}_{\kappa,\mathrm{Y}}(\{\mathsf{var}(\mathrm{Z})\ r\}) = \mathcal{I}_{\kappa,\mathrm{Y}\cup\mathrm{Z}}(r)$
   - $\mathcal{I}_{\kappa,\mathrm{Y}}(\{\mathsf{free}(\mathrm{Z})\ r\}) = \mathcal{I}_{\kappa,\mathrm{Y}\setminus\mathrm{Z}}(r)$
   - $\mathcal{I}_{\kappa,\mathrm{Y}}(r\,@b) = \mathcal{I}_{b,\mathrm{Y}}(r)$

We define the semantics of a formula with respect to an h-word $\mathfrak{w}$ and a tuple $\tau = \langle \kappa, \mathrm{Y}, \upsilon \rangle$ where κ is a Boolean expression indicating the clock context, $\mathrm{Y} \subseteq \mathcal{V}$ is a set of local variables indicating the controlled variables, and $\upsilon \in \{\mathrm{S}, \mathrm{N}, \mathrm{W}\}$ denotes the view (where $\mathrm{S}, \mathrm{N}, \mathrm{W}$ correspond to the *strong*, *neutral* and *weak* views, respectively). For a view υ we use $\overline{\upsilon}$ to denote the *dual view* which is S if $\upsilon = \mathrm{W}$, W if $\upsilon = \mathrm{S}$, and υ otherwise (if $\upsilon = \mathrm{N}$). We use j and k below to denote non-negative integers.

Intuitively, the semantics of local variables are such that an LTL operator does not change the value of a local variable. For example, in the formula $\mathsf{G}\,(var(z)\ \{req, z{\leftarrow}tag\} \mapsto \mathsf{X!}\,\mathsf{F}(ack{\wedge}tag = z))$, the value of z "seen" by the operand of the F operator should be the value assigned by the left-hand side of the suffix implication (exactly as in the following alternative formulation of the same property: $\mathsf{G}\,(var(z)\ \{req, z{\leftarrow}tag\} \mapsto \{\mathsf{true}{\cdot}\mathsf{true}[*]{\cdot}ack \wedge tag = z\}!)$). Conceptually, this means that a formula should "see" the value of the local variables as they were at the beginning of the word, unless a SERE is encountered, in which case the values "seen" should be those dictated by the assignments made within the SERE. We accomplish this as follows: The semantics of $w \models \varphi$ for a word w over Σ takes an initial context of local variables and holds it constant across an extended word $\mathfrak{w}$ whose σ component is exactly w. The semantics of $r!$, r and $r \mapsto \varphi$ then release the constant value of local

variables at the points on the word where a "match" of r is being checked. This allows the local variables to be controlled by r. Finally, in the case of $r \mapsto \varphi$, "after" a match of r has been "seen", the semantics constrains the values of the local variables to take on constant values starting from the end of the "match" of the SERE. We accomplish this using three auxiliary definitions, below.

Definition 23 (Good) *We say that* $\mathfrak{w}$ *is* good *if* $\mathfrak{w} = \epsilon$ *or* $\mathfrak{w} = \mathfrak{a}$ *or* $\mathfrak{w} = \mathfrak{a}\mathfrak{b}\mathfrak{v}$ *and* $\mathfrak{b}\mathfrak{v}$ *is good and* $\mathfrak{b}|_\gamma = \mathfrak{a}|'_\gamma$*, that is, the pre-value of the local variables on a letter is the same as the post-value on the previous letter (i.e. is the result of the assignments that took place on the previous letter).*

Definition 24 (The equivalence class of $\mathfrak{a}$ releasing γ) *Let* $\mathfrak{a} = (\mathfrak{h}, \mathfrak{w})$*. Then* $[\![\mathfrak{a}]\!]$ *denotes the equivalence class* $\{(\mathfrak{h}', \mathfrak{w}') \mid \mathfrak{h}' = \mathfrak{h}$*,* $\mathfrak{w}'$ *is good,* $\mathfrak{w}'|_\sigma = \mathfrak{w}|_\sigma$ *and either* $\mathfrak{w} = \mathfrak{w}' = \epsilon$ *or* $\mathfrak{w}^0|_\gamma = \mathfrak{w}'^0|_\gamma\}$*.*

Definition 25 (Constraining γ on $\mathfrak{a}$) *Let* $\mathfrak{a} = (\mathfrak{h}, \mathfrak{w})$ *and* $\hat{\gamma} \in \Gamma$*. Then* $\mathfrak{a}[(\gamma,\gamma') \leftarrow \hat{\gamma}]$ *denotes the word* $\mathfrak{a}' = (\mathfrak{h}', \mathfrak{w}')$ *such that* $\mathfrak{h}' = \mathfrak{h}$*,* $\mathfrak{w}'|_\sigma = \mathfrak{w}|_\sigma$ *and* $\mathfrak{w}'|_\gamma = \mathfrak{w}'|_{\gamma'} \in \{\hat{\gamma}\}^\infty$*.*

The relation $\mathfrak{a} \models^\tau \varphi$, read "$\mathfrak{a}$ models φ in the context of τ", is formally defined as follows.

Definition 26 (Satisfaction on h-words)

I. $\mathfrak{a} \models^\tau r! \iff \exists \mathfrak{B} \in [\![\mathfrak{a}]\!]$ *s.t. either* $\exists j$ *s.t.* $\mathfrak{B}^{0..j} \in \mathcal{L}_{\kappa,\mathrm{Y}}(r)$ *or* $(\upsilon = \mathrm{W}$ *and* $\mathfrak{B} \in \mathcal{F}_{\kappa,\mathrm{Y}}(r) \cup \mathbb{\lambda})$

II. $\mathfrak{a} \models^\tau r \iff \exists \mathfrak{B} \in [\![\mathfrak{a}]\!]$ *s.t. either* $\exists j$ *s.t.* $\mathfrak{B}^{0..j} \in \mathcal{L}_{\kappa,\mathrm{Y}}(r)$ *or* $\mathfrak{B} \in \mathcal{I}_{\kappa,\mathrm{Y}}(r)$ *or* $(\upsilon \in \{\mathrm{W}, \mathrm{N}\}$ *and* $\mathfrak{B} \in \mathcal{F}_{\kappa,\mathrm{Y}}(r) \cup \mathbb{\lambda})$

III. $\mathfrak{a} \models^\tau r \mapsto \varphi \iff \forall j\ \forall \mathfrak{B} \in [\![\mathfrak{a}]\!]$ *s.t.* $\mathfrak{B}^{0..j} \in \mathcal{L}_{\kappa,\mathrm{Y}}(r)$ *we have that* $\mathfrak{B}^{j\cdot\cdot}[(\gamma,\gamma') \leftarrow \mathfrak{B}^j|_{\gamma'}] \models^\tau \varphi$ *and either* $\upsilon \in \{\mathrm{W}, \mathrm{N}\}$ *or* $w \notin \mathcal{F}_{\kappa,\mathrm{Y}}(r) \cup \{\epsilon\}$

IV. $\mathfrak{a} \models^\tau (\varphi) \iff \mathfrak{a} \models^\tau \varphi$

V. $\mathfrak{a} \models^\tau \neg\varphi \iff \mathfrak{a} \not\models^{\overline{\tau}} \varphi$ *where* $\overline{\tau} = \tau[\upsilon \leftarrow \overline{\upsilon}]$

VI. $\mathfrak{a} \models^\tau \varphi \wedge \psi \iff \mathfrak{a} \models^\tau \varphi$ *and* $\mathfrak{a} \models^\tau \psi$

VII. $\mathfrak{a} \models^\tau \mathsf{X!}[k]\,\varphi \iff$ *either* $\exists j$ *s.t.* $\mathfrak{a}^{0..j} \in \mathcal{L}_{\kappa,\mathrm{Y}}(\mathit{tick}\,[k{+}1])$ *and* $\mathfrak{a}^{j\cdot\cdot} \models^\upsilon \varphi$ *or* $(\upsilon = \mathrm{W}$ *and* $\mathfrak{a} \in \mathcal{F}_{\kappa,\mathrm{Y}}(\mathit{tick}\,[k{+}1]) \cup \mathbb{\lambda})$

VIII. $\mathfrak{a} \models^\tau \varphi \mathsf{U} \psi \iff \exists k$ *s.t.* $\mathfrak{a} \models^\tau \mathsf{X!}[k]\,\psi$ *and* $\forall j < k,\ \mathfrak{a} \models^\tau \mathsf{X!}[j]\,\varphi$

IX. $\mathfrak{a} \models^\tau \varphi$ *sync_abort* $t \iff$ *either* $\mathfrak{a} \models^\tau \varphi$ *or* $\exists k$ *s.t.* $\mathfrak{a}^{k\cdot\cdot} \models^{\tau'} \mathit{ended}(t)$ *and* $\mathfrak{a}^{0..k-1} \models^{\tau''} \varphi$ *where* $\tau' = \tau[\upsilon \leftarrow \mathrm{S}]$ *and* $\tau'' = \tau[\upsilon \leftarrow \mathrm{W}]$

X. $\mathfrak{a} \models^\tau (\mathsf{var}(\mathrm{Z})\ \varphi) \iff \mathfrak{a} \models^{\tau'} \varphi$ *where* $\tau' = \tau[\mathrm{Y} \leftarrow \mathrm{Y} \cup \mathrm{Z}]$

XI. $\mathfrak{a} \models^\tau \varphi @ b \iff \mathfrak{a} \models^{\tau'} \varphi$ *where* $\tau' = \tau[\kappa \leftarrow b]$

Definition 27 (Satisfaction over Σ) *Let* w *be a word over* Σ *and* φ *be an FL formula. We say that* w models φ *in the view* υ*, denoted* $w \models^\upsilon \varphi$ *iff for all* $\hat{\gamma} \in \Gamma$ *and all extended words* $\mathfrak{w}$ *such that* $\mathfrak{w}|_\sigma = w$ *we have that* $(\epsilon, \mathfrak{w})[(\gamma,\gamma') \leftarrow \hat{\gamma}] \models^{\tau_\upsilon} \varphi$*, where* $\tau_\upsilon = \langle \mathsf{true}, \emptyset, \upsilon \rangle$*. That is, the past is empty, the pre- and post-values are* γ *on every letter, the initial clock context is* true *and the initial set of controlled variables is empty.*

Definition 28 (Holds Weakly, Strong, Neutrally, Pending, Fails) *Let* w *be a word over* Σ *and* φ *an FL formula. We say that*

- φ holds weakly *on* w *if* $w \models^{\mathrm{W}} \varphi$
- φ holds (neutrally) *on* w *if* $w \models^{\mathrm{N}} \varphi$
- φ holds strongly *on* w *if* $w \models^{\mathrm{S}} \varphi$
- φ *is* pending *on word* w *iff* $w \models^{\mathrm{W}} \varphi$ *and* $w \not\models^{\mathrm{N}} \varphi$
- φ fails *on word* w *iff* $w \not\models^{\mathrm{W}} \varphi$

B.3.2 Semantics of OBE formulas

The semantics of OBE formulas are defined over states in the *model*, rather than finite or infinite words. Let f be an OBE formula, $M = (S, S_0, R, \mathcal{P}, L)$ a model and $s \in S$ a state of the model. The notation $M, s \models f$ means that f holds in state s of model M. The notation $M \models f$ is equivalent to $\forall s \in S_0 : M, s \models f$. In other words, f is valid for every initial state of M.

The semantics of OBE formulas are defined inductively, using as the base case the semantics of *Basic Boolean expressions* over *letters* in $2^{\mathcal{P}}$, as given in Definition 7.

The semantics of an OBE formula are those of standard CTL. The semantics are defined as follows, where b denotes a Boolean expression and f, f_1, and f_2 denote OBE formulas.

1. $M, s \models b \iff L(s) \in \mathcal{B}(b = \mathrm{T})$
2. $M, s \models (f) \iff M, s \models f$
3. $M, s \models \neg f \iff M, s \not\models f$
4. $M, s \models f_1 \wedge f_2 \iff M, s \models f_1$ and $M, s \models f_2$
5. $M, s \models \mathsf{EX}\ f \iff$ there exists a computation path π of M such that $|\pi| > 1$, $\pi_0 = s$, and $M, \pi_1 \models f$
6. $M, s \models \mathsf{E}[f_1\ \mathsf{U}\ f_2] \iff$ there exists a computation path π of M such that $\pi_0 = s$ and there exists $k < |\pi|$ such that $M, \pi_k \models f_2$ and for every j such that $j < k$: $M, \pi_j \models f_1$
7. $M, s \models \mathsf{EG}\ f \iff$ there exists a computation path π of M such that $\pi_0 = s$ and for every j such that $0 \leq j < |\pi|$: $M, \pi_j \models f$

B.4 Syntactic Sugaring

The remainder of the temporal layer is syntactic sugar. In other words, it does not add expressive power, and every piece of syntactic sugar can be defined in terms of the basic operators presented above. The syntactic sugar is defined below.

NOTE –
The definitions given here do not necessarily represent the most efficient implementation. In some cases, there is an equivalent syntactic sugaring, or a direct implementation, that is more efficient.

B.4.1 Additional Boolean expressions

Let $b \in \mathcal{B}$ and $e \in \mathcal{E}$. Then additional Boolean expressions can be viewed as abbreviations of the core Boolean expressions given in Definition 1, as follows:

1. $rose(b) \stackrel{\text{def}}{=} \neg prev(b) \wedge b$
2. $fell(b) \stackrel{\text{def}}{=} prev(b) \wedge \neg b$
3. $stable(e) \stackrel{\text{def}}{=} prev(e) = e$

B.4.2 Additional expressions

Let $e \in \mathcal{E}$. Then additional expressions can be viewed as abbreviations of the expressions given in Definition 2, as follows:

– $prev(e) \stackrel{\text{def}}{=} prev(e, 1)$

B.4.3 Additional SEREs

We regard a *procedural block* as mechanism for both calculating a set of expressions and assigning them to a set of local variables which are given to the procedural block as inputs. Formally,

– $f(z_1, \ldots, z_n)$ where f is a procedural block and $z_1, \ldots, z_n$ are local variables is interpreted as an abbreviation for the sequence of assignments $z_1 \leftarrow e_{f_1}, \ldots, z_n \leftarrow e_{f_n}$ where $e_{f_1}, \ldots, e_{f_n}$ are expressions corresponding to intermediate calculations of f.

Let $b, c \in \mathcal{B}$ and $e \in \mathcal{E}$. Let k be a non-negative integer. Then the following are SEREs:

1. $rose(b, c) \stackrel{\text{def}}{=} rose(b)@c$
2. $fell(b, c) \stackrel{\text{def}}{=} fell(b)@c$
3. $stable(e, c) \stackrel{\text{def}}{=} stable(e)@c$
4. $prev(e, k, c) \stackrel{\text{def}}{=} prev(e, k)@c$
5. $ended(r, c) \stackrel{\text{def}}{=} ended(r)@c$

B.4.4 Additional SERE operators

Let i, j, k, and l be integer constants such that $i \geq 0$, $j \geq i$, $k \geq 1$, $l \geq k$. Then additional SERE operators can be viewed as abbreviations of the core SERE operators given in Definition 3, as follows, where b denotes a boolean expression, and r denotes a SERE.

1. $r[*] \stackrel{\text{def}}{=} [*0] \cup r[+]$
2. $r[*0] \stackrel{\text{def}}{=} [*0]$
3. $r[*k] \stackrel{\text{def}}{=} \overbrace{r \cdot r \cdot \ldots \cdot r}^{k \ times}$
4. $r[*i..j] \stackrel{\text{def}}{=} r[*i] \mid \ldots \mid r[*j]$
5. $r[*i..] \stackrel{\text{def}}{=} r[*i] \cdot r[*]$
6. $r[*..i] \stackrel{\text{def}}{=} r[*0] \mid \ldots \mid r[*i]$
7. $r[*..] \stackrel{\text{def}}{=} r[*0..]$
8. $[+] \stackrel{\text{def}}{=} \mathsf{true}[+]$
9. $[*] \stackrel{\text{def}}{=} \mathsf{true}[*]$
10. $[*i] \stackrel{\text{def}}{=} \mathsf{true}[*i]$
11. $[*i..j] \stackrel{\text{def}}{=} \mathsf{true}[*i..j]$
12. $[*i..] \stackrel{\text{def}}{=} \mathsf{true}[*i..]$
13. $[*..i] \stackrel{\text{def}}{=} \mathsf{true}[*..i]$
14. $[*..] \stackrel{\text{def}}{=} \mathsf{true}[*..]$
15. $b[= i] \stackrel{\text{def}}{=} \{\neg b[*] \cdot b\}[*i] \cdot \neg b[*]$

16. $b[= i..j] \stackrel{\text{def}}{=} b[= i] \mid ... \mid b[= j]$
17. $b[= i..] \stackrel{\text{def}}{=} b[= i]{\cdot}[*]$
18. $b[= ..i] \stackrel{\text{def}}{=} b[= 0] \mid ... \mid b[= i]$
19. $b[= ..] \stackrel{\text{def}}{=} b[= 0..]$
20. $b[\rightarrow] \stackrel{\text{def}}{=} \neg b[*]{\cdot}b$
21. $b[\rightarrow k] \stackrel{\text{def}}{=} \{\neg b[*]{\cdot}b\}[*k]$
22. $b[\rightarrow k..l] \stackrel{\text{def}}{=} b[\rightarrow k] \mid ... \mid b[\rightarrow l]$
23. $b[\rightarrow k..] \stackrel{\text{def}}{=} b[\rightarrow k] \mid \{b[\rightarrow k]{\cdot}[*]{\cdot}b\}$
24. $b[\rightarrow ..k] \stackrel{\text{def}}{=} b[\rightarrow 1] \mid ... \mid b[\rightarrow k]$
25. $b[\rightarrow ..] \stackrel{\text{def}}{=} b[\rightarrow 1..]$
26. $\mathsf{skip} \stackrel{\text{def}}{=} \{\mathsf{free}(\mathcal{V})\ \mathsf{true}\}$ where $\mathcal{V}$ is the entire set of local variables
27. $r_1\ \&\ r_2 \stackrel{\text{def}}{=} \{\ \{r_1{\cdot}\mathsf{skip}[*]\} \cap r_2\ \} \cup \{\ r_1 \cap \{r_2{\cdot}\mathsf{skip}[*]\}\ \}$
28. $r_1\ within\ r_2 \stackrel{\text{def}}{=} \{\mathsf{skip}[*]{\cdot}\ r_1{\cdot}\ \mathsf{skip}[*]\}\ \cap\ \{r_2\}$
29. $\{\mathsf{var}(\mathrm{Z}{\leftarrow}\mathrm{E})\ r\} \stackrel{\text{def}}{=} \{\mathsf{var}(\mathrm{Z})\ \{\{\mathsf{true}, \mathrm{Z} \leftarrow \mathrm{E}\}{\circ}r\} \cup \{[*0]\ \cap\ r\}\}$

B.4.5 Additional FL operators

Let i, j, k and l are integers such that $i \geq 0$, $j \geq i$, $k > 0$ and $l \geq k$. Then additional FL operators can be viewed as abbreviations of the core operators given in Definition 4, as follows, where b denotes a boolean expression, r, r_1, and r_2 denote SEREs, and φ, φ_1, and φ_2 denote FL formulas.

1. $\varphi_1 \vee \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2)$
2. $\varphi_1 \rightarrow \varphi_2 \stackrel{\text{def}}{=} \neg\varphi_1 \vee \varphi_2$
3. $\varphi_1 \leftrightarrow \varphi_2 \stackrel{\text{def}}{=} (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$

4. $\mathsf{X}[i]\varphi \stackrel{\text{def}}{=} \neg\mathsf{X!}[i]\ \neg\varphi$
5. $\mathsf{X!}\varphi \stackrel{\text{def}}{=} \mathsf{X!}[1]\ \varphi$
6. $\mathsf{X}\varphi \stackrel{\text{def}}{=} \mathsf{X}[1]\ \varphi$

7. $\mathsf{F}\varphi \stackrel{\text{def}}{=} \mathsf{true\ U}\ \varphi$
8. $\mathsf{G}\varphi \stackrel{\text{def}}{=} \neg\mathsf{F}\neg\varphi$
9. $\varphi_1\ \mathsf{W}\ \varphi_2 \stackrel{\text{def}}{=} (\varphi_1\ \mathsf{U}\ \varphi_2) \vee \mathsf{G}\varphi_1$

10. $always\ \varphi \stackrel{\text{def}}{=} \mathsf{G}\ \varphi$
11. $never\ \varphi \stackrel{\text{def}}{=} \mathsf{G}\ \neg\varphi$

12. $next!\ \varphi \stackrel{\text{def}}{=} \mathsf{X}!\ \varphi$
13. $next\ \varphi \stackrel{\text{def}}{=} \mathsf{X}\ \varphi$
14. $eventually!\ \varphi \stackrel{\text{def}}{=} \mathsf{F}\varphi$

15. $\varphi_1\ until!\ \varphi_2 \stackrel{\text{def}}{=} \varphi_1\ \mathsf{U}\ \varphi_2$
16. $\varphi_1\ until\ \varphi_2 \stackrel{\text{def}}{=} \varphi_1\ \mathsf{W}\ \varphi_2$
17. $\varphi_1\ until!\_\ \varphi_2 \stackrel{\text{def}}{=} \varphi_1\ \mathsf{U}\ (\varphi_1 \wedge \varphi_2)$
18. $\varphi_1\ until\_\ \varphi_2 \stackrel{\text{def}}{=} \varphi_1\ \mathsf{W}\ (\varphi_1 \wedge \varphi_2)$

19. $\varphi_1\ before!\ \varphi_2 \stackrel{\text{def}}{=} (\neg\varphi_2)\ \mathsf{U}\ (\varphi_1 \wedge \neg\varphi_2)$
20. $\varphi_1\ before\ \varphi_2 \stackrel{\text{def}}{=} (\neg\varphi_2)\ \mathsf{W}\ (\varphi_1 \wedge \neg\varphi_2)$
21. $\varphi_1\ before!\_\ \varphi_2 \stackrel{\text{def}}{=} (\neg\varphi_2)\ \mathsf{U}\ \varphi_1$
22. $\varphi_1\ before\_\ \varphi_2 \stackrel{\text{def}}{=} (\neg\varphi_2)\ \mathsf{W}\ \varphi_1$

23. $next![i]\ \varphi \stackrel{\text{def}}{=} \mathsf{X}![i]\ \varphi$
24. $next[i]\ \varphi \stackrel{\text{def}}{=} \mathsf{X}[i]\ \varphi$
25. $next\_a![i..j]\varphi \stackrel{\text{def}}{=} (\mathsf{X}![i]\varphi) \wedge \ldots \wedge (\mathsf{X}![j]\varphi)$
26. $next\_a[i..j]\varphi \stackrel{\text{def}}{=} (\mathsf{X}[i]\varphi) \wedge \ldots \wedge (\mathsf{X}[j]\varphi)$
27. $next\_e![i..j]\varphi \stackrel{\text{def}}{=} (\mathsf{X}![i]\varphi) \vee \ldots \vee (\mathsf{X}![j]\varphi)$
28. $next\_e[i..j]\varphi \stackrel{\text{def}}{=} (\mathsf{X}[i]\varphi) \vee \ldots \vee (\mathsf{X}[j]\varphi)$

29. $next\_event!(b)(\varphi) \stackrel{\text{def}}{=} (\neg b)\ \mathsf{U}\ b \wedge \varphi$
30. $next\_event(b)(\varphi) \stackrel{\text{def}}{=} (\neg b)\ \mathsf{W}\ (b \wedge \varphi)$
31. $next\_event!(b)[k](\varphi) \stackrel{\text{def}}{=} next\_event!(b)\ \overbrace{(\mathsf{X}!\ next\_event!(b)...(\mathsf{X}!\ next\_event!(b)}^{k-1\ times}(\varphi))...)$
32. $next\_event(b)[k](\varphi) \stackrel{\text{def}}{=} next\_event(b)\overbrace{(\mathsf{X}next\_event(b)...(\mathsf{X}next\_event(b)}^{k-1\ times}(\varphi))...)$
33. $next\_event\_a!(b)[k..l](\varphi) \stackrel{\text{def}}{=} next\_event!(b)[k](\varphi) \wedge ... \wedge next\_event!(b)[l](\varphi)$
34. $next\_event\_a(b)[k..l](\varphi) \stackrel{\text{def}}{=} next\_event(b)[k](\varphi) \wedge ... \wedge next\_event(b)[l](\varphi)$
35. $next\_event\_e!(b)[k..l](\varphi) \stackrel{\text{def}}{=} next\_event!(b)[k](\varphi) \vee ... \vee next\_event!(b)[l](\varphi)$
36. $next\_event\_e(b)[k..l](\varphi) \stackrel{\text{def}}{=} next\_event(b)[k](\varphi) \vee ... \vee next\_event(b)[l](\varphi)$

37. $r(\varphi) \stackrel{\text{def}}{=} r \mapsto \varphi$

38. $r \mapsto \varphi \stackrel{\text{def}}{=} \{r \circ \{\mathsf{true}\cdot\mathsf{true}\}\} \mapsto \varphi$

39. φ async_abort $t \stackrel{\text{def}}{=} \varphi$ sync_abort $(t@\mathsf{true})$

40. φ abort $t \stackrel{\text{def}}{=} \varphi$ async_abort t

B.4.6 Parameterized SEREs and Formulas

Let r be a SERE, and l, m integers. Let S be a set of constants, integers or boolean values and p an identifier. The left hand side of the following are SEREs, equivalent to the SEREs on the right hand side.

1. for p *in* S : $\cup$ $r \stackrel{\text{def}}{=} \bigcup_{s\in S}\{r[p \leftarrow s]\}$

2. for $p\langle l..m\rangle$ *in* S : $\cup$ $r \stackrel{\text{def}}{=} \bigcup_{s_l\in S}\ldots\bigcup_{s_m\in S}\{r[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]\}$

3. for p *in* S : $\cap$ $r \stackrel{\text{def}}{=} \bigcap_{s\in S}\{r[p \leftarrow s]\}$

4. for $p\langle l..m\rangle$ *in* S : $\cap$ $r \stackrel{\text{def}}{=} \bigcap_{s_l\in S}\ldots\bigcap_{s_m\in S}\{r[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]\}$

5. for p *in* S : $\&$ $r \stackrel{\text{def}}{=} \&_{s\in S}\{r[p \leftarrow s]\}$

6. for $\&$ $p\langle l..m\rangle$ *in* S : $\&$ $r \stackrel{\text{def}}{=} \&_{s_l\in S}\ldots\&_{s_m\in S}\{r[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]\}$

Where $r[p \leftarrow s]$ is the SERE obtained from r by replacing every occurrence of p by s and $r[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]$ is the SERE obtained from r by replacing every occurrence of p_j with s_j for all j such that $l \le j \le m$.

Let φ be an FL formula, and l, m integers. Let S be a set of constants, integers or boolean values and p an identifier. The left hand side of the following are FL formulas equivalent to the FL formulas on the right hand side.

1. for p *in* $S : \vee\ \varphi \stackrel{\text{def}}{=} \bigvee_{s\in S}\varphi[p \leftarrow s]$

2. for $p\langle l..m\rangle$ *in* $S : \vee\ \varphi \stackrel{\text{def}}{=} \bigvee_{s_l\in S}\ldots\bigvee_{s_m\in S}\varphi[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]$

3. for p *in* $S : \wedge\ \varphi \stackrel{\text{def}}{=} \bigwedge_{s\in S}\varphi[p \leftarrow s]$

4. for $p\langle l..m\rangle$ *in* $S : \wedge\ \varphi \stackrel{\text{def}}{=} \bigwedge_{s_l\in S}\ldots\bigwedge_{s_m\in S}\varphi[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]$

5. forall p *in* S : $\varphi \stackrel{\text{def}}{=}$ for p *in* $S : \wedge\ \varphi$

6. forall $p\langle l..m\rangle$ *in* S : $\varphi \stackrel{\text{def}}{=}$ for $p\langle l..m\rangle$ *in* $S : \wedge\ \varphi$

where $\varphi[p \leftarrow s]$ is the formula obtained from φ by replacing every occurrence of p by s and $\varphi[p\langle l..m\rangle \leftarrow \langle s_l..s_m\rangle]$ is the formula obtained from φ by replacing every occurrence of p_j with s_j for all j such that $l \le j \le m$.

B.4.7 Additional OBE operators

Let f, f_1, f_2 denote OBE formulas. Then additional OBE operators can be derived from the core OBE operators given in Definition 5 as follows:

1. $f_1 \vee f_2 = \neg(\neg f_1 \wedge \neg f_2)$
2. $f_1 \rightarrow f_2 = \neg f_1 \vee f_2$
3. $f_1 \leftrightarrow f_2 = (f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_1)$
4. $\mathsf{EF} f = \mathsf{E}[\mathsf{true}\ \mathsf{U}\ f]$
5. $\mathsf{AX} f = \neg\mathsf{EX}\neg f$
6. $\mathsf{A}[f_1\ \mathsf{U}\ f_2] = \neg(\mathsf{E}[\neg f_2\ \mathsf{U}\ (\neg f_1 \wedge \neg f_2)] \vee \mathsf{EG}\neg f_2)$
7. $\mathsf{AG} f = \neg\mathsf{E}[\mathsf{true}\ \mathsf{U}\ \neg f]$
8. $\mathsf{AF} f = \mathsf{A}[\mathsf{true}\ \mathsf{U}\ f]$

B.5 Rewriting rules for clocks

In Section B.3.1.2 we gave the semantics of clocked FL formulas directly. There is an equivalent definition in terms of FL formulas without clocks, as follows: Starting from the outermost clock, use the following rules to translate clocked SEREs into unclocked SEREs, and clocked FL formulas into unclocked FL formulas.

The rewrite rules for SEREs are:

1. $\mathcal{R}^c(\{r\}) = \mathcal{R}^c(r)$
2. $\mathcal{R}^c(b) = \neg c[*] \cdot (c \wedge b)$
3. $\mathcal{R}^c(b, \mathrm{Z}{\leftarrow}\mathrm{E}) = \neg c[*] \cdot (c \wedge b, \mathrm{Z}{\leftarrow}\mathrm{E})$
4. $\mathcal{R}^c(r_1 \cdot r_2) = \mathcal{R}^c(r_1) \cdot \mathcal{R}^c(r_2)$
5. $\mathcal{R}^c(r_1 \circ r_2) = \{\mathcal{R}^c(r_1)\} \circ \{\mathcal{R}^c(r_2)\}$
6. $\mathcal{R}^c(r_1\ \cup\ r_2) = \{\mathcal{R}^c(r_1)\}\ \cup\ \{\mathcal{R}^c(r_2)\}$
7. $\mathcal{R}^c(r_1\ \cap\ r_2) = \{\mathcal{R}^c(r_1)\}\ \cap\ \{\mathcal{R}^c(r_2)\}$
8. $\mathcal{R}^c([*0]) = [*0]$
9. $\mathcal{R}^c(r[+]) = \{\mathcal{R}^c(r)\}[+]$
10. $\mathcal{R}^c(\mathsf{var}(\mathrm{Z})\ r) = \{\mathsf{var}(\mathrm{Z})\ \mathcal{R}^c(r)\}$
11. $\mathcal{R}^c(\mathsf{free}(\mathrm{Z})\ r) = \{\mathsf{free}(\mathrm{Z})\ \mathcal{R}^c(r)\}$
12. $\mathcal{R}^c(r@c_1) = \mathcal{R}^{c_1}(r)$

The rewrite rules for FL formulas are:

1. $\mathcal{F}^c((\varphi)) = (\mathcal{F}^c(\varphi))$

2. $\mathcal{F}^c(\neg\varphi) = \neg\mathcal{F}^c(\varphi)$

3. $\mathcal{F}^c(\varphi \wedge \psi) = (\mathcal{F}^c(\varphi) \wedge \mathcal{F}^c(\psi))$

4. $\mathcal{F}^c(r!) = \mathcal{R}^c(r)!$

5. $\mathcal{F}^c(r) = \mathcal{R}^c(r)$

6. $\mathcal{F}^c(r \mapsto \varphi) = \mathcal{R}^c(r) \mapsto \mathcal{F}^c(\varphi)$

7. $\mathcal{F}^c(\mathsf{X!}[0]\varphi) = (\neg c\ \mathsf{U}\ (c \wedge \mathcal{F}^c(\varphi))))$
   $\mathcal{F}^c(\mathsf{X!}\varphi) = (\neg c\ \mathsf{U}\ (c \wedge \mathsf{X!}\ (\neg c\ \mathsf{U}\ (c \wedge \mathcal{F}^c(\varphi)))))$
   $\mathcal{F}^c(\mathsf{X!}[k]\varphi) = \mathcal{F}^c(\underbrace{\mathsf{X!} \ldots \mathsf{X!}}_{k\ times}\varphi)$ for $k \geq 2$

8. $\mathcal{F}^c(\varphi\ \mathsf{U}\ \psi) = ((c \rightarrow \mathcal{F}^c(\varphi))\ \mathsf{U}\ (c \wedge \mathcal{F}^c(\psi)))$

9. $\mathcal{F}^c(\varphi\ \mathsf{sync\_abort}\ t) = \mathcal{F}^c(\varphi)\ \mathsf{sync\_abort}\ \mathcal{F}^c(t)$

10. $\mathcal{F}^c(\mathsf{var}(\mathrm{Z})\ \varphi) = (\mathsf{var}(\mathrm{Z})\ \mathcal{F}^c(\varphi))$

11. $\mathcal{F}^c(\varphi @ c_1) = \mathcal{F}^{c_1}(\varphi)$

Annex C

(informative)

Bibliography

[B1] Eisner, C., Fisman, D., Havlicek, J., Yoad Lustig, A. M., and Campenhout, D. V., Reasoning with temporal logic on truncated paths. In *Proc. 15th International Conference on Computer-Aided Verification conference (CAV'O3)*, volume 2725 of *LNCS*, pages 27–39, 2003.

[B2] Eisner, C., Fisman, D., Havlicek, J., McIsaac, A., and Campenhout, D. V., The definition of a temporal clock operator. In *ICALP*, volume 2719 of *Lecture Notes in Computer Science*, pages 857–870. Springer, 2003.

[B3] Eisner, C., Fisman, D., Havlicek, J., and Martensson, J., The *\top, \bottom* approach to truncated semantics. Technical Report 2006.01, Accellera, 2006.

[B4] Eisner, C., Fisman, D., Havlicek, J., A topological characterization of weakness. In *PODC '05: Proceedings of the twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–8, New York, NY, USA, 2005. ACM.

[B5] Eisner, C., Fisman, D., Proposal for extending Annex B of PSL with local variables, procedural blocks, past expressions and clock alignment operators. Technical Report H-0256, IBM, 2008.

[B6] Eisner, C., Fisman, D., Structural Contradictions. In Proc. 4th International *Haifa Verification Conference*, volume 5394 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2008.

[B7] Eisner, C., Fisman, D., Augmenting a Regular Expression-Based Temporal Logic with Local Variables. FMCAD'08: Proc. 8th International Conference on Formal Methods in Computer-Aided Design, 2008.

[B8] Fisman, D., On the characterization of **until** as a fixed point under clocked semantics. In Proc. International *Haifa Verification Conference*, volume 4899 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2007.

[B9] Havlicek, J., Fisman, D., and Eisner, C., Basic Results on the Semantics of Accellera PSL1.1. Technical Report 2004.02, Accellera, 2004.

[B10] Manna, Z., and Pnueli, A., *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[B11] Pnueli, A., A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

Index

