IEC 62531

Edition 1.0 2007-11

INTERNATIONAL STANDARD

IEEE 1850™

Standard for Property Specification Language (PSL)

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

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 62531-2007

THIS PUBLICATION IS COPYRIGHT PROTECTED

Copyright © 2007 IEEE

All rights reserved. IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Inc.

Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the IEC Central Office.

Any questions about IEEE copyright should be addressed to the IEEE. Enquiries about obtaining additional rights to this publication and other information requests should be addressed to the IEC or your local IEC member National Committee.

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

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

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

- Catalogue of IEC publications: www.iec.ch/searchpub
The IEC on-line Catalogue enables you to search by a variety of criteria (reference number, text, technical committee,…). It also gives information on projects, withdrawn and replaced publications.

- IEC Just Published: www.iec.ch/online_news/justpub
Stay up to date on all new IEC publications. Just Published details twice a month all new publications released. Available on-line and also by email.

- Electropedia: www.electropedia.org
The world's leading online dictionary of electronic and electrical terms containing more than 20 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary online.

- Customer Service Centre: www.iec.ch/webstore/custserv
If you wish to give us your feedback on this publication or need further assistance, please visit the Customer Service Centre FAQ or contact us:

Email: csc@iec.ch
Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 62531-2007

IEC 62531

Edition 1.0 2007-11

IEEE 1850™

INTERNATIONAL STANDARD

Standard for Property Specification Language (PSL)

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

PRICE CODE XG

ICS 25.040

ISBN 2-8318-9352-6

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 62531-2007

CONTENTS

FOREWORD 4
IEEE introduction 7
1. Overview 8
1.1 Scope 8
1.2 Purpose 8
1.2.1 Background 8
1.2.2 Motivation 9
1.2.3 Goals 9
1.3 Usage 9
1.3.1 Functional specification 9
1.3.2 Functional verification 10
2. Normative references 14
3. Definitions, acronyms, and abbreviations 16
3.1 Definitions 16
3.2 Acronyms and abbreviations 19
3.3 Special terms 19
4. Organization 22
4.1 Abstract structure 22
4.1.1 Layers 22
4.1.2 Flavors 22
4.2 Lexical structure 23
4.2.1 Identifiers 23
4.2.2 Keywords 23
4.2.3 Operators 24
4.2.4 Macros 29
4.2.5 Comments 31
4.3 Syntax 31
4.3.1 Conventions 31
4.3.2 HDL dependencies 32
4.4 Semantics 36
4.4.1 Clocked vs. unclocked evaluation 36
4.4.2 Safety vs. liveness properties 37
4.4.3 Linear vs. branching logic 37
4.4.4 Simple subset 37
4.4.5 Finite-length vs. infinite-length behavior 38
4.4.6 The concept of strength 38
5. Boolean layer 40
5.1 Expression type classes 40
5.1.1 Bit expressions 40
5.1.2 Boolean expressions 41
5.1.3 BitVector expressions 42
5.1.4 Numeric expressions 42
5.1.5 String expressions 43
5.2 Expression forms 43
5.2.1 HDL expressions 43

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

5.2.2 PSL expressions ... 45
5.2.3 Built-in functions ... 45
5.2.4 Union expressions ... 51
5.3 Clock expressions ... 51
5.4 Default clock declaration ... 53

6. Temporal layer ... 56

6.1 Sequential expressions ... 57
6.1.1 Sequential Extended Regular Expressions (SEREs) ... 57
6.1.2 Sequences ... 64
6.2 Properties ... 70
6.2.1 FL properties ... 71
6.2.2 Optional Branching Extension (OBE) properties ... 91
6.2.3 Replicated properties ... 98
6.3 Property and sequence declarations ... 100
6.3.1 Parameters ... 101
6.3.2 Declarations ... 103
6.3.3 Instantiation ... 104

7. Verification layer ... 108

7.1 Verification directives ... 108
7.1.1 assert ... 108
7.1.2 assume ... 109
7.1.3 assume_guarantee ... 110
7.1.4 restrict ... 110
7.1.5 restrict_guarantee ... 111
7.1.6 cover ... 112
7.1.7 fairness and strong_fairness ... 112
7.2 Verification units ... 113
7.2.1 Verification unit binding ... 114
7.2.2 Verification unit inheritance ... 116
7.2.3 Verification unit scoping rules ... 117

8. Modeling layer ... 120

8.1 Integer ranges ... 120
8.2 Structures ... 121
Annex A (normative) Syntax rule summary ... 122
Annex B (normative) Formal syntax and semantics of IEEE Std 1850 PSL ... 136
Annex C (informative) Bibliography ... 146
Annex D (informative) List of participants ... 148

Index ... 150

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Documents
Document Sharing Hub
Full document details: IEEE 62531-2007

INTERNATIONAL ELECTROTECHNICAL COMMISSION

STANDARD FOR PROPERTY SPECIFICATION LANGUAGE (PSL)

FOREWORD

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

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

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

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

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

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

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

IEC/IEEE Dual Logo International Standards

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

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

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

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

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

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

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

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

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

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

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

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Documents
Document Sharing Hub
Full document details: IEEE 62531-2007

IEEE Standard for
Property Specification Language (PSL)

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

and the
IEEE Standards Association Corporate Advisory Group

Approved 22 September 2005

IEEE-SA Standards Board

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

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

GDL: General Description Language, Accellera, Mar. 2005

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

Documents
Document Sharing Hub

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 62531-2007

IEEE Introduction

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

Notice to users

Errata

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

Interpretations

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

Patents

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

STANDARD FOR
PROPERTY SPECIFICATION VOLTAGE (PSL)

1. Overview

1.1 Scope

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

1.2 Purpose

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

1.2.1 Background

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

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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.

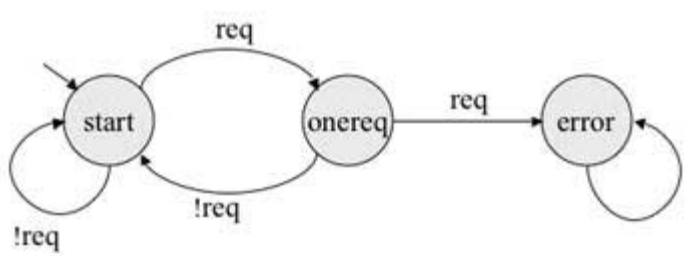

Figure 1—A simple (deterministic) FSM that checks Property 1

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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.

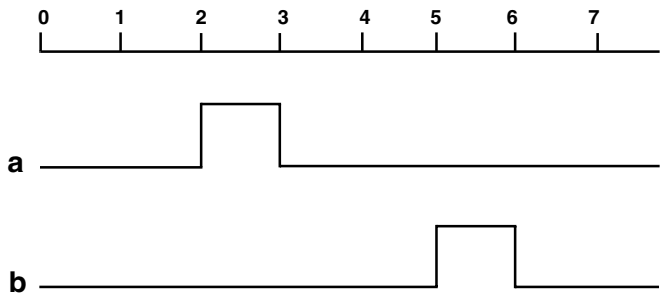

Figure 2—A trace that satisfies Property 2

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.

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

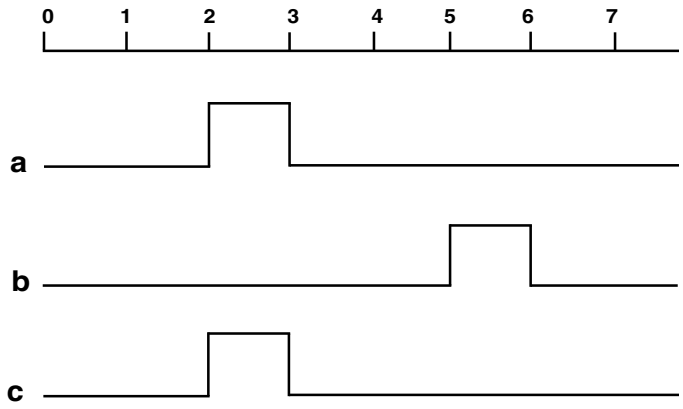

Figure 3—A trace that satisfies Property 3

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*.

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Documents
Document Sharing Hub
Full document details: IEEE 62531-2007

2. Normative references

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

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

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

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

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

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

IEEE P1666™, Draft Standard for the SystemC Language.

IEEE P1800™, Draft Standard for the SystemVerilog Language.

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

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

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

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

3. Definitions, acronyms, and abbreviations

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

3.1 Definitions

This subclause defines the terms used in this standard.

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

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

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

3.1.4 behavior: A path.

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

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

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

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

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

3.1.10 count: A number or range.

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

3.1.12 cycle: An evaluation cycle.

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

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

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

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1.29 model checking: A type of formal verification.

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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

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

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

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

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

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

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

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

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

3.1.39 range: A series of consecutive numbers, from a low bound to a high bound, inclusive, such that the low bound is less than or equal to the high bound. In particular, this includes the case in which the low bound is equal to the high bound. Also, a pair of statically computable integer expressions specifying such a series of consecutive numbers, where the left expression specifies the low bound of the series, and the right expression specifies the high bound of the series. A range may describe a set of values or a variable number of cycles or event repetitions.

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

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

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

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

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

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

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

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

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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

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

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

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

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

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

3.2 Acronyms and abbreviations

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

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

3.3 Special terms

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

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

— **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.

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

4. Organization

4.1 Abstract structure

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

4.1.1 Layers

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

4.1.1.1 Boolean layer

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

4.1.1.2 Temporal layer

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

4.1.1.3 Verification layer

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

4.1.1.4 Modeling layer

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

4.1.2 Flavors

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

4.1.2.1 SystemVerilog flavor

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

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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

4.1.2.2 Verilog flavor

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

4.1.2.3 VHDL flavor

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

4.1.2.4 SystemC flavor

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

4.1.2.5 GDL flavor

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

4.2 Lexical structure

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

4.2.1 Identifiers

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

Example

```
mutex
Read_Transaction
L_123
```

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

4.2.2 Keywords

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

The keywords used in PSL are shown in Table 1.

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

Table 1—Keywords

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

Table 2—FL operator precedence and associativity

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

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

4.2.3.2.1 Union operator

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

`union` union operator

The union operator is left-associative.

4.2.3.2.2 Clocking operator

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

`@` clock event

The clocking operator is left-associative.

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

4.2.3.2.3 SERE repetition operators

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

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

SERE repetition operators are left-associative.

4.2.3.2.4 Sequence within operator

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

`within`	sequence within operator

The sequence within operator is left-associative.

4.2.3.2.5 Sequence conjunction operators

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

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

Sequence conjunction operators are left-associative.

4.2.3.2.6 Sequence disjunction operator

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

`|`	sequence disjunction

The sequence disjunction operator is left-associative.

4.2.3.2.7 Sequence fusion operator

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

`:`	sequence fusion

The sequence fusion operator is left-associative.

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

4.2.3.2.8 Sequence concatenation operator

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

`;`	sequence concatenation

The sequence concatenation operator is left-associative.

4.2.3.2.9 FL termination operators

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

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

The FL termination operators are left-associative.

4.2.3.2.10 FL occurrence operators

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

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

FL occurrence operators are right-associative.

4.2.3.2.11 Bounding operators

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

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

FL bounding operators are right-associative.

4.2.3.2.12 Suffix implication operators

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

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

The suffix implication operators are right-associative.

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

4.2.3.2.13 Logical implication operators

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

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

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

4.2.3.2.14 FL invariance operators

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

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

FL occurrence operators are right-associative.

4.2.3.3 Optional Branching Extension (OBE) operators

Table 3—OBE operator precedence and associativity

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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:

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

```
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
```

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Documents
Document Sharing Hub
Full document details: IEEE 62531-2007

4.2.5 Comments

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

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

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

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

4.3 Syntax

4.3.1 Conventions

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

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

   PSL_Statement

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

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

   vunit **(** **;**

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

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

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

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

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

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

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

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

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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.

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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

Syntax 4-2—Flavor macro HDL_UNIT

4.3.2.2 HDL_DECL and HDL_STMT

PSL verification units may contain certain kinds of HDL declarations and statements. Flavor macros `HDL_DECL` and `HDL_STMT` connect the PSL syntax with the syntax for declarations and statements in the grammar for each HDL. Both of these are shown in Syntax 4-3.

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

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

Syntax 4-3—Flavor macros HDL_DECL and HDL_STMT

4.3.2.3 HDL_EXPR and HDL_CLOCK_EXPR

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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

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

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

Syntax 4-4—Flavor macro HDL_EXPR and HDL_CLOCK_EXPR

4.3.2.4 HDL_VARIABLE_TYPE

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

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

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

Syntax 4-5—Flavor macro HDL_VARIABLE_TYPE

4.3.2.5 HDL_RANGE

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

```
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*.

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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.

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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.

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

— 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.

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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

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

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

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

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

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

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

4.4.6.2 Strong satisfaction

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

4.4.6.3 Relating the two concepts of strength

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

5. Boolean layer

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

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

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

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

5.1 Expression type classes

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

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

```
Any_Type ::=
  HDL_or_PSL_Expression
```

Syntax 5-11—Any type expression

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

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

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

5.1.1 Bit expressions

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

```
Bit ::=
  bit_HDL_or_PSL_Expression
```

Syntax 5-12—Bit expression

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

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

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

In SystemC, types sc_bit and sc_bv are Bit types.

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

5.1.2 Boolean expressions

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

```
Boolean ::=
  boolean_HDL_or_PSL_Expression
```

Syntax 5-13—Boolean expression

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

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

In SystemC, type bool is a Boolean type.

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

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

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

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

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

5.1.3 BitVector expressions

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

```
BitVector ::=
  bitvector_HDL_or_PSL_Expression
```

Syntax 5-14—BitVector expression

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

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

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

5.1.4 Numeric expressions

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

```
Number ::=
  numeric_HDL_or_PSL_Expression
```

Syntax 5-15—Numeric expression

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

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

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

Documents
Document Sharing Hub
Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

Restrictions

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

5.1.5 String expressions

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

```
String ::=
   string_HDL_or_PSL_Expression
```

Syntax 5-16—String expression

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

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

5.2 Expression forms

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

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

Syntax 5-17—HDL or PSL Expression

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

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

5.2.1 HDL expressions

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

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

```
HDL_Expression ::=
  HDL_EXPR

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

Syntax 5-18—HDL expression

Informal Semantics

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

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

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

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

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

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

See 7.2 for an explanation of verification units and modes.

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007

5.2.2 PSL expressions

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

```
HDL_or_PSL_Expression ::=
   PSL_Expression

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

Syntax 5-19—PSL expression

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

Informal Semantics

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

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

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

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

5.2.3 Built-in functions

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

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

Syntax 5-20—Built-in functions

Published by IEC under licence from IEEE. © 2005 IEEE. All rights reserved.
Authorized licensed use limited to: Tsinghua University. Downloaded on February 25,2026 at 21:02:01 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 62531-2007