IEEE STANDARDS ASSOCIATION

IEEE Standard for the Functional Verification Language *e*

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

REDLINE
Shows exact changes from the previous version!

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

26 August 2011

IEEE Std 1647™-2011
(Revision of
IEEE Std 1647-2008)

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

IEEE STANDARDS ASSOCIATION

This is a Redline Document produced by Techstreet, a business of Thomson Reuters.

This document is intended to provide users with an indication of changes from one edition to the next. It includes a full-text version of the new document, plus an indication of changes from the previous version.

Redlines are designed to save time and improve efficiencies by using the latest software technology to find and highlight document changes. More professionals are using valuable new technologies like redlines, to help improve outcomes in a fastpaced global business world.

Because it may not be technically possible to capture all changes accurately, it is recommended that users consult previous editions as appropriate. In all cases, only the current base version of this publication is to be considered the official document.

445 Hoes Lane, Piscataway, NJ 08854 USA | http://standards.ieee.org | Tel. +1 732-981-0060 Fax +1 732-562-1571

Documents Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

Redline Processing Notes:

1. ~~**Red Text**~~ **-** Red strikethrough text denotes deletions.

2. **Blue Text** **-** Blue underlined text denotes modifications and additions.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

IEEE Std 1647™-2011
(Revision of
IEEE Std 1647-2008)

IEEE Standard for the Functional Verification Language *e*

Sponsor

Design Automation Standards Committee
of the
IEEE Computer Society

Approved 16 June 2011
IEEE SA-Standards Board

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

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

Namespaces for Types in ***e***, Version 1.1

e Language Reference, Version 5.1, Chapter 26 (Encapsulation Constructs)

e Language Reference, Version 9.2, Chapter 6 (Template Types), Chapter 12 (Generation Constraints), Chapter 21 (Macros)

e Language Reference, Version 10.2, Chapter 2 (***e*** Data Types), Chapter 6 (***e*** Ports)

e Reuse Methodology Developer Manual, Version 2.1, Chapter 5 (Sequences), Chapter 6 (Messaging)

e Ports, Chapter 6

Specman Beta Features, Chapter 4 (Constant Fields and Constant when Subtypes), Chapter 15 (Reflection Interface for ***e***)

Abstract: The ***e*** *functional verification language* is an application-specific programming language, aimed at automating the task of verifying a hardware or software design with respect to its specification. Verification environments written in ***e*** provide a model of the environment in which the design is expected to function, including the kinds of erroneous conditions the design needs to withstand. A typical verification environment is capable of generating user-controlled test inputs with statistically interesting characteristics. Such an environment can check the validity of the design responses. Functional coverage metrics are used to control the verification effort and gauge the quality of the design. ***e*** verification environments can be used throughout the design cycle, from a high-level architectural model to a fully realized system. A definition of the ***e*** language syntax and semantics and how tool developers and verification engineers should use them are contained in this standard.

Keywords: assertion, concurrent programming, constraint, dynamic verification, functional coverage, functional verification, IEEE 1647, simulation, temporal logic, test generation

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

Copyright © 2011 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 26 August 2011. Printed in the United States of America.

IEEE and POSIX are registered trademarks in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Incorporated.

Java is a trademark of Sun Microsystems, Inc. in the United States and other countries.

Perl is a registered trademark of Perl, Inc.

SystemC is a registered trademark of Open SystemC Initiative.

Verilog is a registered trademark of Cadence Design Systems, Inc.

PDF: ISBN 978-0-7381-6687-2 STD97128
Print: ISBN 978-0-7381-6688-9 STDPD97128

IEEE prohibits discrimination, harassment and bullying. For more information, visit http://www.ieee.org/web/aboutus/whatis/policies/p9-26.html. No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

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

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

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

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation, or every ten years for stabilization. When a document is more than five years old and has not been reaffirmed, or more than ten years old and has not been stabilized, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon his or her independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration. A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal interpretation of the IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Recommendations to change the status of a stabilized standard should include a rationale as to why a revision or withdrawal is required. Comments and recommendations on standards, and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
Piscataway, NJ 08854-4141
USA

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.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

Introduction

This introduction is not part of IEEE Std 1647-2011, IEEE Standard for the Functional Verification Language ***e***.

The ***e*** *functional verification language* is an application-specific programming language aimed at the problem of verifying functional correctness of hardware and software designs. Simply stated, functional verification attempts to provide a quantitative answer to the question: How well does the design match the functional specification?

Functional correctness of chip designs grew in criticality from the mid-1980s. As design complexity kept growing, ad hoc testing methods ran out of steam and a more systematic verification approach was necessary. Manually constructed test suites, the early method of choice, became both uneconomical and ineffective when scaled up. As a result, many companies supplemented manual test suites with *pseudo-random generation* of input stimulus. Such test generation programs were typically built for a particular project or a particular architecture. They turned out to be expensive to develop and maintain, but once functional, they would clean up the design in a very thorough way.

A key observation made by Yoav Hollander, the creator of ***e***, was that verification environments of different projects have a lot in common and yet each verification environment is structured to match a particular design specification. Hollander's solution was to create a language that had verification-specific constructs as primitives and the full capabilities of a high-level language for customization. In particular, pseudo-random test generation became a built-in capability of the language. Early prototypes of the language were experimented with as early as 1993, showing significant productivity gains.

The ***e*** language was productized by Verisity, Ltd., in 1996, as part of a functional verification tool suite. The proliferation of the ***e*** language and the growing investment in ***e***-based intellectual property (IP) compelled the creation of the ***e*** *steering committee* in June of 2002, composed of individuals from Texas Instruments, Rambus, STMicroelectronics, Cisco, Intel, Axis System, STARC, and Verisity. The ***e*** *steering committee* recommended the ***e*** language be standardized through the Institute of Electrical and Electronics Engineers (IEEE). Accepting the recommendation, Verisity released the rights to the language to the IEEE in June of 2003.

The ***e*** language, in its current form, brings together concepts from many domains.

— ***e*** has a basic object-oriented (OO) programming model, with implicit memory management and single inheritance. In this, ***e*** is similar to Java™.[a]

— Beyond objects, ***e*** supports *aspects*, which can be viewed as layers cutting across multiple objects. Adding an aspect to an existing program refines the program by introducing a coherent change to a plurality of objects.

— ***e*** supports constraints as object features. Constraints are used to refine object modeling. The execution model of the language involves resolving constraint systems and picking random values that would satisfy constraint systems.

— ***e*** is a strongly typed language, like Pascal and Modula.

— ***e*** has concurrency constructs and modeling blocks for hierarchical composition, similar to hardware description languages (HDLs) like Verilog® (see IEEE Std 1364™) and VHSIC hardware description language (VHDL) (see IEC/IEEE 61691-1-1).[a, b] Concurrency in ***e*** is synchronous, like in Esterel.

— ***e*** contains a temporal language that borrows from Linear Temporal Logic and Interval Temporal Logic.

[a]Java is a trademark of Sun Microsystems, Inc. in the United States and other countries. Verilog is a registered trademark of Cadence Design Systems, Inc. This information is given for the convenience of users of this standard and does not constitute an endorsement by the IEEE of these products. Equivalent products may be used if they can be shown to lead to the same results.

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

Copyright © 2011 IEEE. All rights reserved.

Documents Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

— ***e*** has many built-in constructs aimed at simplifying common programming tasks; built-in support for lists and hashes; and pattern matching and string- and file-manipulation features, which are borrowed from Perl®.[c]

— The ***e*** syntax is extendable with a powerful macro capability.

This combination of concepts caters directly to the needs of verification engineers, removing the need to cobble together multiple components in different languages.

As with any programming language, the source of ingenuity is with the programmer. Verification engineers need sound methodologies, creativity, an inquisitive mind, and a keen eye for poorly specified aspects. Yet experience with ***e*** shows that when put to good use, the ***e*** language fosters productivity and quality results.

Notice to users

Laws and regulations

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

Copyrights

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

Updating of IEEE documents

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

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

[c]Perl is a registered trademark of Perl, Inc. This information is given for the convenience of users of this standard and does not constitute an endorsement by the IEEE of these products. Equivalent products may be used if they can be shown to lead to the same results.

Copyright © 2011 IEEE. All rights reserved.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

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. 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, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses. Other Essential Patent Claims may exist for which a statement of assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Copyright © 2011 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

Participants

At the time this standard was submitted to the IEEE-SA Standards Board for approval, the IEEE ***e*** Functional Verification Language (***e***WG) Working Group had the following members and observers:

Darren Galpin, *Chair*
Amy Witherow, *Technical Editor*

Charu Aggarwal
Munnu Bajpai
Victor Berman
Parag Bhatt
Martin J. Bishop
Danila Chernetsov
Kai Moon Chow
Keith Chow
S. Claassen
Jonathan David
Sandeep Desai
Thomas Dineen
Sourav Dutta
Allan Gillard
James Gray
Randall Groves
M. Hashmi
Werner Hoelzl
Joseph Hupcey III
Piotr Karocki
Asad Khan
Stanley Krolikoski
Brett G. Lammers
Jim Lewis
Adam Ley
Daniel Lindberg
Greg Luri
Francoise Martinolle
Michael T. McNamara
William Milam
Robert Myers
Andrew Piziali
Ulrich Pohl
Iulian Profir
Robert Robinson
Bartien Sayogo
Gil Shultz
Mark Strickland
Walter Struppler
Matan Vax
Srinivasa Vemuru
David G. Von Bank
Henry J. Von Bank
Jovan Rade Vujnic
Janusz Zalewski

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

Christoffer Amlo
Hugh Barrass
Michael Benjamin
Victor Berman
Parag Bhatt
Avi Bloch
Danila Chernetsov
Keith Chow
Tommy Cooper
Jacob Daniel
Sandeep Desai
Stylianos Diamantidis
Thomas Dineen
George Economakos
James Gilb
Sergiu Goma
Randall Groves
M. Hashmi
Werner Hoelzl
Dennis Horwitz
Joseph Hupcey III
Piotr Karocki
Jake Karrfalt
Yaron Kashai
Eran Keydar
Zeev Kirshenbaum
Brett G. Lammers
Jim Lewis
Guoliang Liu
William Lumpkins
Greg Luri
Chris Macionski
Riccardo Mariani
Arthur Marris
Michael T. McNamara
Gary Michel
Michael Mirmak
Robert Myers
Michael S. Newman
William Paulsen
Steven Pearlmutter
Gregory Peterson
Robert Peterson
Andrew Piziali
Vikram Punj
William Schilp
John Shields
Lori Kate Smith
Michael Stellfox
Mark Strickland
Ajayharsh Varikat
Matan Vax
Srinivasa Vemuru
David G. Von Bank
Henry J. Von Bank
Paul Work

Copyright © 2011 IEEE. All rights reserved.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

When the IEEE-SA Standards Board approved this standard on 16 June 2011, it had the following membership:

Richard H. Hulett, *Chair*
John Kulick, *Vice Chair*
Robert M. Grow, *Past Chair*
Judith Gorman, *Secretary*

Masayuki Ariyoshi
William Bartley
Ted Burse
Clint Chaplin
Wael Diab
Jean-Philippe Faure
Alexander Gelman
Paul Houzé
Jim Hughes
Joseph L. Koepfinger*
David J. Law
Thomas Lee
Hung Ling
Oleg Logvinov
Ted Olsen
Gary Robinson
Jon Walter Rosdahl
Sam Sciacca
Mike Seavey
Curtis Siller
Phil Winston
Howard L. Wolfman
Don Wright

*Member Emeritus

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

Satish K. Aggarwal, *NRC Representative*
Richard DeBlasio, *DOE Representative*
Michael Janezic, *NIST Representative*

Don Messina
IEEE Standards Program Manager, Document Development

Joan M. Woolery
IEEE Standards Program Manager, Technical Program Development

Copyright © 2011 IEEE. All rights reserved.
Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

Contents

1. Overview ... 1

1.1 Scope ... 1
1.2 Purpose ... 1
1.3 Verification environments ... 1
1.4 Basic concepts relating to this standard ... 2
1.5 Conventions used ... 8
1.6 Use of color in this standard ... 9
1.7 Contents of this standard ... 10

2. Normative references ... 13

3. Definitions, acronyms, and abbreviations ... 13

3.1 Definitions ... 13
3.2 Acronyms and abbreviations ... 15

4. *e* basics ... 17

4.1 Lexical conventions ... 17
4.2 Syntactic elements ... 24
4.3 Struct hierarchy and name resolution ... 29
4.4 Ranges ... 36
4.5 Operator precedence ... 36
4.6 Evaluation order of expressions ... 36
4.7 Bitwise operators ... 38
4.8 Boolean operators ... 39
4.9 Arithmetic operators ... 41
4.10 Comparison operators ... 42
4.11 String matching ... 46
4.12 Extraction and concatenation operators ... 48
4.13 Scalar modifiers ... 52
4.14 Parentheses ... 53
4.15 list.method() ... 53
4.16 Special-purpose operators ... 54

5. Data types ... 59

5.1 e data types ... 59
5.2 Untyped expressions ... 64
5.3 Assignment rules ... 65
5.4 Real data type ... 68
5.5 Precision rules for numeric operations ... 70
5.6 Automatic type casting ... 72
5.7 Defining and extending scalar types ... 73
5.8 Type-related constructs ... 75

6. Structs, subtypes, and fields ... 81

6.1 Structs overview ... 81
6.2 Defining structs: struct ... 82

Copyright © 2011 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

6.3 Extending structs: extend type ... 83
6.4 Restrictions on inheritance ... 83
6.5 Extending subtypes ... 84
6.6 Creating subtypes with when ... 84
6.7 Extending when subtypes ... 86
6.8 Defining fields: field ... 87
6.9 Defining list fields ... 89
6.10 Projecting list of fields ... 91
6.11 Defining attribute fields ... 91

7. Units ... 93

7.1 Overview ... 93
7.2 Defining units and fields of type unit ... 95
7.3 Unit attributes ... 99
7.4 Predefined methods of any_unit ... 100
7.5 Unit-related predefined methods of any_struct ... 102
7.6 Unit-related predefined routines ... 104

8. Template types ... 107

8.1 Defining a template type ... 107
8.2 Instantiating a template type ... 109
8.3 Template types and reflection ... 110
8.4 Template errors ... 111
8.5 Limitations ... 112
8.6 Templates versus macros ... 112

9. ***e*** ports ... 115

9.1 Introduction to ***e*** ports ... 115
9.2 Using simple ports ... 116
9.3 Using buffer ports ... 118
9.4 Using event ports ... 119
9.5 Using method ports ... 120
9.6 Defining and referencing ports ... 122
9.7 Port attributes ... 127
9.8 Buffer port methods ... 139
9.9 MVL methods for simple ports ... 141
9.10 Global MVL routines ... 147
9.11 Comparative analysis of ports and tick access ... 150
9.12 ***e*** port binding ... 151
9.13 TLM interface ports in ***e*** ... 152

10. Constraints and generation ... 161

10.1 Types of constraints ... 161
10.2 Generation concepts ... 161
10.3 Type constraints ... 175
10.4 Defining constraints ... 177
10.5 Invoking generation ... 185

Copyright © 2011 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

11. Events ... 187

11.1 Causes of events ... 187
11.2 Scope of events ... 187
11.3 Defining and emitting named events ... 187
11.4 Predefined events ... 188

12. Temporal expressions ... 191

12.1 Overview ... 191
12.2 Temporal operators and constructs ... 194
12.3 Success and failure of a temporal expression ... 209

13. Temporal struct members ... 211

13.1 on ... 211
13.2 expect | assume ... 212

14. Time-consuming actions ... 213

14.1 Synchronization actions ... 213
14.2 Concurrency actions ... 214
14.3 State machines ... 216

15. Coverage constructs ... 221

15.1 Defining coverage groups: cover ... 221
15.2 Defining basic coverage items: item ... 223
15.3 Defining cross coverage items: cross ... 227
15.4 Defining transition coverage items: transition ... 229
15.5 Extending coverage groups: cover ... using also ... is also ... 230
15.6 Extending coverage items: item ... using also ... 231
15.7 Coverage API ... 232
15.8 Coverage methods for the covers struct ... 238

16. Macros ... 243

16.1 Overview ... 243
16.2 define-as statement ... 244
16.3 define-as-computed statement ... 244
16.4 Match expression structure ... 245
16.5 Interpretation of match expressions ... 248
16.6 Macro expansion code ... 249

17. Print, checks, and error handling ... 253

17.1 print ... 253
17.2 Handling DUT errors ... 253
17.3 Handling user errors ... 258
17.4 Handling programming errors: assert ... 260

Copyright © 2011 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

18. Methods 261

18.1 Rules for defining and extending methods 261
18.2 Invoking methods 269
18.3 Parameter passing 272
18.4 Using the C interface 274

19. Creating and modifying ***e*** variables 277

19.1 About ***e*** variables 277
19.2 var 277
19.3 = ... 278
19.4 op= 278
19.5 <= 279

20. Packing and unpacking 281

20.1 Basic packing 281
20.2 Predefined pack options 284
20.3 Customizing pack options 285
20.4 Packing and unpacking specific types 285
20.5 Implicit packing and unpacking 291

21. Control flow actions 293

21.1 Conditional actions 293
21.2 Iterative actions 295
21.3 File iteration actions 299
21.4 Actions for controlling the program flow 300

22. Importing and preprocessor directives 303

22.1 Importing ***e*** modules 303
22.2 #ifdef, #ifndef 304
22.3 #define 305
22.4 #undef 306

23. Encapsulation constructs 307

23.1 package: package-name 307
23.2 package: type-declaration 307
23.3 package | protected | private: struct-member 308
23.4 Scope operator (::) 309

24. Simulation-related constructs 311

24.1 force 311
24.2 release 311
24.3 Tick access: 'hdl-pathname' 312
24.4 simulator_command() 312
24.5 stop_run() 313

Copyright © 2011 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

25. Messages ... 315

25.1 Overview ... 315
25.2 The message model ... 315
25.3 message and messagef ... 316
25.4 Message loggers ... 318
25.5 Configuring message loggers with constraints ... 319
25.6 Messaging procedural interface ... 320
25.7 Examples ... 328

26. Sequences ... 331

26.1 Overview ... 331
26.2 Sequence statement ... 333
26.3 do sequence action ... 335
26.4 Sequence struct types and members ... 336
26.5 BFM-driver-sequence flow diagrams ... 341

27. List pseudo-methods library ... 345

27.1 Pseudo-methods overview ... 345
27.2 Using list pseudo-methods ... 345
27.3 Pseudo-methods to modify lists ... 345
27.4 General list pseudo-methods ... 354
27.5 Math and logic pseudo-methods ... 368
27.6 List CRC pseudo-methods ... 370
27.7 Keyed list pseudo-methods ... 372

28. Predefined methods library ... 375

28.1 Predefined methods of sys ... 375
28.2 Predefined methods of any_struct ... 375
28.3 Methods and predefined attributes of unit any_unit ... 378
28.4 Pseudo-methods ... 378
28.5 Coverage methods ... 380

29. Predefined routines library ... 381

29.1 Deep copy and compare routines ... 381
29.2 Integer arithmetic routines ... 384
29.3 Real arithmetic routines ... 388
29.4 bitwise_op() ... 389
29.5 get_all_units() ... 390
29.6 String routines ... 390
29.7 Output routines ... 399
29.8 Operating system interface routines ... 401
29.9 set_config() ... 404
29.10 Random routines ... 404

Copyright © 2011 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

30. Predefined file routines library 407

30.1 File names and search paths 407
30.2 File handles 407
30.3 Low-level file methods 407
30.4 General file routines 412
30.5 Reading and writing structs 418

31. Reflection API 421

31.1 Introduction 421
31.2 Type information 422
31.3 Aspect information 431
31.4 Value query and manipulation 435

32. Predefined resource sharing control structs 439

32.1 Semaphore methods 439
32.2 How to use the semaphore struct 440

33. IP protection 445

33.1 Encryption 445
33.2 Decryption 445
33.3 Reflection API 446
33.4 Encryption Targets 446

Annex A (informative) Bibliography 447

Annex B (normative) Source code serialization 449

Annex C (informative) Comparison of when and like inheritance 457

Annex D (normative) Name spaces 465

Annex E (informative) Reflection API examples 473

Annex F (informative) Encryption Targets 477

Copyright © 2011 IEEE. All rights reserved.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

IEEE Standard for the Functional Verification Language *e*

IMPORTANT NOTICE: This standard is not intended to ensure safety, security, health, or environmental protection ~~in all circumstances~~. Implementers of the standard are responsible for determining appropriate safety, security, environmental, and health practices or regulatory requirements.

This IEEE document is made available for use subject to important notices and legal disclaimers. These notices and disclaimers appear in all publications containing this document and may be found under the heading "Important Notice" or "Important Notices and Disclaimers Concerning IEEE Documents." They can also be obtained on request from IEEE or viewed at http://standards.ieee.org/IPR/ disclaimers.html.

1. Overview

This clause explains the scope and purpose of this standard; gives an overview of the basic concepts, major semantic components, and conventions used in this standard; and summarizes its contents.

1.1 Scope

~~The scope of this~~This standard ~~is the definition of~~defines the *e* functional verification language. This standard aims to serve as an authoritative source for the definition of

(a) syntax and semantics of *e* language constructs

(b) the *e* language interaction with standard simulation languages,

and

(c) *e* language libraries

~~This revision extends the standard to cover novel verification-related features~~.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

1.2 Purpose

This standard serves the community involved with functional verification of electronic designs using the ***e*** language. It provides an implementation independent definition of the ***e*** language and facilitates the development of ***e*** language based design automation tools. ~~The revision project extends the standard to include novel verification related features.~~

1.3 Verification environments

Electronic systems are integrated circuits (ICs), boards, or modules combining multiple ICs together, along with optional embedded processors and software components. Electronic systems are built to *specifications* that anticipate the environment in which such systems are expected to function and define the expected system functionality. *Functional verification* measures how well a system meets its specification. Even with moderately complex systems this question cannot be answered by inspection. For all modern electronic systems, a sophisticated *verification process* needs to accompany the design process to ensure compliance with the specification.

Many electronic design automation (EDA) tools are used to carry out the functional verification process. The most prominent functional verification method, used to verify virtually all system designs, is called *dynamic verification* or *simulation-based verification*. Simulation-based verification makes use of a functional model of the system being designed. The functional model is *simulated* in the context of a mock- up of the anticipated system environment. This mock-up is called the *verification environment*.

There are many requirements a verification environment needs to fulfill, as follows:

— It needs to create input stimulus and feed it into the system being verified.

— It needs to collect the output from the system being verified, as well as the state of selected internal nodes.

— It needs to check that the output matches the expectations, based on the functional requirements, the state of the system being verified, and the inputs provided.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

— It needs to measure *functional coverage*: the extent to which functions of the system being verified have been exercised by the verification environment.

— It needs to facilitate error identification, isolation, and debug. For that purpose, test environments contain combinational and temporal assertions, as well as various messaging and logging capabilities.

— The verification environment needs to be able to mimic all possible variations and configurations the system being verified might face in practice.

— The verification environment needs to be able to throw a wide variety of error conditions at the system being verified, in order to test error handling and error recovery.

— The verification environment should be easily controllable, to allow steering by the verification engineers.

The verification environment is a primary component in a simulation-based verification process. The environment needs to drive the system being verified through enough diverse scenarios to cover a statistically meaningful portion of the systems state space. Coverage data collected throughout the process should provide the foundation to an informed decision about the production readiness of the system being designed.

Sophisticated verification environments are complex software systems, representing a significant investment. Reuse of verification components is a primary way of minimizing this investment. Reusability is typically an artifact of a well thought-out software architecture, but in the case of ***e***, the language itself facilitates reuse through aspect-oriented programming (AOP) constructs and the semantics of generation.

The ***e*** functional verification language can facilitate the creation of sophisticated verification environments, as ***e*** features many constructs that automate and support common verification environment tasks.

A standard definition of the ***e*** language should serve both practicing verification engineers and EDA tool developers. Engineers using ***e*** to build verification environments and reusable verification components need to ensure the valuable intellectual property (IP) they create can be interpreted by others. Tool developers need to agree on consistent syntax and semantics to ensure interoperability between tools. These goals are best facilitated by means of an open standard.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

1.4 Basic concepts relating to this standard

This subclause discusses some basic concepts relating to the ***e*** functional verification language. ***e*** is built upon many of the concepts shared by most programming languages—these concepts are not repeated here. The concepts in the following subclauses are either unique to ***e*** or contain some specific details pertaining to ***e***.

1.4.1 Fundamental considerations

An ***e*** *program* is a computer program written in the ***e*** functional verification language. ***e*** is a *Turing complete* programming language.[1] While aimed specifically at constructing functional verification environments, ***e*** can be used to create arbitrary programs.

In principle, an ***e*** program can be either compiled or interpreted. A compiled execution flow requires the availability of an ***e*** *compiler*, which reads in the ***e*** program and produces a machine-executable image of the program. Such machine-executable images can execute, or run, on suitable computers. Alternatively, an ***e*** program can be loaded into an ***e*** *interpreter* and execute within the interpreter context. A computer program executing ***e*** code in either mode shall be called an ***e*** *runtime engine*.

1.4.2 Organization of *e* programs

e programs comprise one or more text files called *modules*. Each module can contain ***e*** code and comments. The syntactic form of code and comments is discussed in Clause 4.

An ***e*** module can require the presence of other modules. Such requirements can be specified explicitly, using the **import** statement, as well as derived automatically. The considerations for importing modules and resolving module dependencies are discussed in Clause 22 and Annex B, respectively. Each ***e*** program induces an *ordered* list of modules. Loading the modules in a different order might result in a program that is malformed or is different from the intended program.

e programs can be interpreted or compiled as a whole. The ***e*** semantics allow, but do not require, an alternative mode where an initial list of modules is loaded or compiled first, and later modules are introduced to the execution environment, in order, during program execution. This "load on top" mode has important practical implications, such as debugging and development efficiency, but it is entirely in the implementation domain.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

1.4.3 Modes of execution

e programs are distinguished based on the modeling of time during execution. A more complete discussion of treatment of time can be found in Clause 11 and Clause 12.

1.4.3.1 Stand-alone execution

e programs can execute independently. In such cases, the program shall use cycle-based semantics. Time shall be modeled by an internal source generating a sequence of discrete time units (*ticks*). The time variable shall be assigned consecutive integer values, starting from 0 (before the initial tick). The advancement of time shall stop when the program requests simulation to stop or for an exception. These conditions are covered under 24.5, 29.9, and Clause 17.

[1]A programming language or any other logical system is called *Turing complete* if it has a computational power equivalent to a universal Turing machine.

1.4.3.2 Co-execution with a master simulator

e programs can execute in tandem with a *master simulator*. In this case, control is passed back and forth between the ***e*** runtime engine and the master simulator. The master simulator is typically a commercial Verilog®, VHDL, or SystemC® simulator that runs a simulation of the system being verified.[2]

In this execution mode, the time semantics are controlled by the master simulator. The executing ***e*** program defines conditions under which execution of the master simulator is suspended and control is passed to the ***e*** runtime engine. These switches of control are called *callbacks*. A more detailed discussion of callback conditions can be found in 11.4.2.

The master simulator defines the representation of time and the time increments, based on its native semantics. For example, an event-driven Verilog simulator determines time representation based on the timescale in effect and the increments of time based on the processing of simulation events.

Upon each callback, the ***e*** execution engine shall read the time value from the master simulator. That time value shall hold for the entire duration of the tick (until control is returned to the master simulator).

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

In a co-execution environment, the ***e*** execution engine can read and write values of entities represented in the master simulator domain. Such values are generally latched at the start of each tick. Changes are applied only at the point control is passed back to the master simulator. Further details on the interaction with foreign entities can be found in Clause 9 and Clause 24.

1.4.4 Execution phases

The execution of ***e*** programs goes through the phases listed in Ta ble 1.

Table 1—Execution phases

Phase name	Purpose	Methods called
init	Initialize objects.	**init()**
setup	Used to hook up verification environment components; configure environment.	**sys.setup()**
generate	Execute an elaboration and initial generation of the object hierarchy rooted in **sys**.	**pre_generate()** **post_generate()**
run	Launch time-consuming methods (TCMs) in the context of an object; execute the timed portion of the program.	**run()**
extract	Collect log and coverage information after execution.	**sys.xtract()**
check	Perform final checking.	**sys.check()**

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

finalize	Clean up prior to termination.	**quit()**

The phases can be customized to meet the needs of particular verification environment tasks. The methods listed can be extended to customize each phase. Methods listed with a **sys.** prefix are only available at the topmost object called **sys**. Other methods are available for any object. Such methods shall be called in order as long as the object can be reached from **sys**.

[2]Verilog is a registered trademark of Cadence Design Systems, Inc. SystemC is a registered trademark of Open SystemC Initiative. This information is given for the convenience of users of this standard and does not constitute an endorsement by the IEEE of these products. Equivalent products may be used if they can be shown to lead to the same results.

For more information about the object instance hierarchy in an ***e*** program, see 4.3.1. The methods defined **any_struct** are discussed in 28.2.

1.4.5 Major semantic components

The rest of this subclause highlights some key components of the ***e*** language.

1.4.5.1 Types

e is a strongly typed language. Every scalar and object created during the execution of an ***e*** program has an associated type. Such scalars and objects are said to be *instances* of their type. Types in ***e*** completely define the features and functions of all their instances.

e features a *single inheritance* scheme. According to this scheme, each type has exactly one parent type. The type hierarchy is rooted in some predefined types for objects and scalars.

The association of instances and types in ***e*** can be *static* or *dynamic*. Static typing means an instance is associated with a particular type at creation and that association does not change during the life of the instance. Dynamic typing means the association between instance and type can change during the lifetime of the instance. Dynamic inheritance (sometimes called *when inheritance*), enables the creation of instances that adapt during their lifetime to changing conditions—changing their features and functions accordingly. Do not confuse dynamic typing with *casting*, where an instance is converted to a type with compatible features. Nor

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

is dynamic typing the same as up-casting: an object-oriented (OO) concept that allows an object to be accessed as one of its super-types, abstracting away some of the type features.

A comprehensive discussion of static and dynamic typing in ***e*** can be found in Annex C.

1.4.5.2 Packages, aspects, and information hiding

e packages comprise ordered collections of modules. Packages are typically used to implement a particular functionality or feature set. Packages are *aspects*: they refine and extend predefined types, and optionally introduce new types.

Information hiding is a fundamental OO concept that isolates an object's published interface from the implementation details. Hiding means that certain features are simply out of scope; there is no way to access them from outside an object. In ***e***, types, as well as type features, can be subject to information hiding.

e introduces two orthogonal relationships in order to implement an AOP viable information hiding scheme: type membership and package membership. Features can be public, private to an object, private to a package, or private to both. Figure 1 depicts these relationships.

Type A and Type B are globally visible. Type C is private to Package Q. X is a globally visible feature of A, and Yis visible within Package P. Zis a feature of B, which is defined in Package P, but is visible anywhere within B. W is a feature of B defined in Package Q and is only visible in the intersection of Package Qand Type B.

For a detailed discussion of packages and information hiding, see Clause 23.

1.4.5.3 Numeric expressions and values

e supports signed and unsigned integer values of arbitrary length. Numbers can be represented as binary, octal, decimal, and hexadecimal values. ***e*** inherits number representation and much of its arithmetic and logical semantics from the C language (see ISO/IEC 9899).[3] Semantics of specific bit-oriented, logical, and arithmetic operators can be found in 4.7, 4.8, and 4.9, respectively.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

Values passed on from external domains, such as a master simulator, are converted to *e* values at the interface, as they are passed through ports. Similarly, *e* values can be converted to drive multi-valued logic at the interface. Value conversion is discussed in 9.9.

e ~~does not support~~supports double-precision floating-point numbers as the real data type.

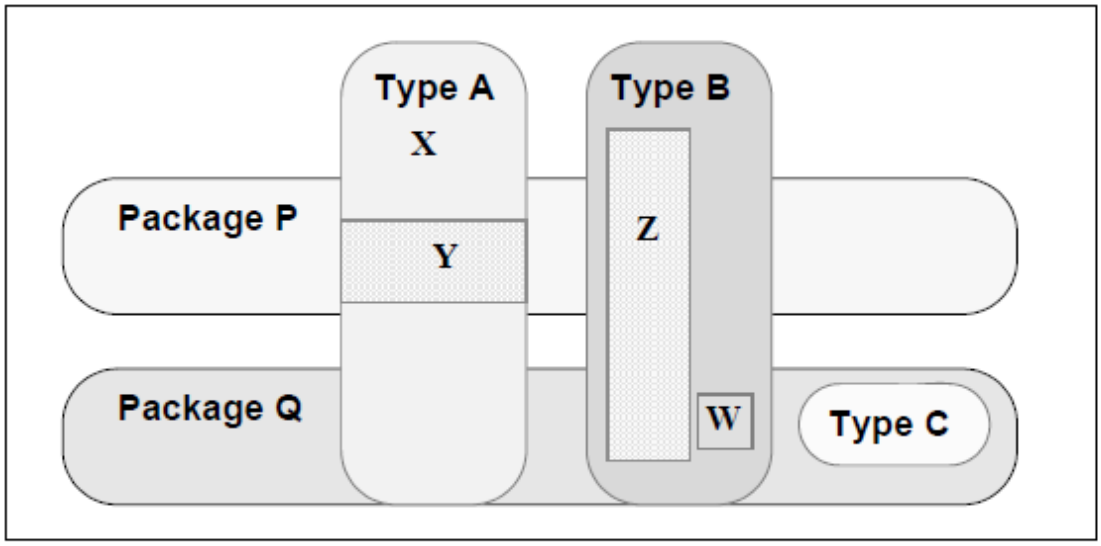

Figure 1—Type and package membership

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

1.4.5.4 Constraints and generation

Constraints in ***e*** are features of object types. Constraints introduce relationships among fields accessible from the object scope or between such fields and values (constants or dynamically computed values). Constraints affect generation actions only; they do not affect imperative assignment. For instance, an integer field might be constrained to a specific value, say 1. Assigning another value to that field is legal. Initialization shall assign 0to that field (the default value for integers). Generation of the field shall assign it to 1, based on the constraint.

Constraints are primarily declarative. Each generation operation involves a closure of types and constraints affecting the operation. The closure defines the *constraint satisfaction problem* (CSP) that needs to be resolved to construct an instance hierarchy and assign values according to the constraints involved. The CSP might have multiple solutions (i.e., different instance hierarchies and/or different value assignments that satisfy all constraints); such CSPs are called *under-constrained*. In such cases, a solution shall be randomly selected from the set of available solutions.

The programmer can use this to define an under-constrained problem, indicating that random variations are allowed. Given this, different implementations may produce different results and still remain compliant with this standard.

[3]For information on references, see Clause 2.

Constraint-based generation is used for the following two main purposes during execution:

— Instantiation of the initial unit tree, hooking up port connections and instantiation of object instances within the unit tree

— Creation of object hierarchies on demand, typically to serve as input stimulus driving the simulation

Constraints and generation are discussed in Clause 10.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

1.4.5.5 Concurrent execution

e semantics support concurrent execution of computation. Concurrent computation can be spawned explicitly, using the **start** action. Implicit concurrency is created by defining temporal struct members, such as **event**, **assume**, and **expect**. See Clause 11 and Clause 14 for a detailed discussion of these constructs.

In software implementations, concurrent execution is simulated by running multiple threads of execution within an operating system (OS) process. The rules that govern time multiplexing between threads of execution are called *threading semantics*. ***e*** mandates that threading is non-preemptive: the running thread can yield control, but control cannot be taken away. Threads in ***e*** yield control when they become blocked. Threads can block when they wait on, or synchronize to, a temporal expression (TE); attempt to perform a port operation that blocks; or attempt to access a shared resource that is occupied. The programmer can assume atomic execution for sequences of actions not containing these operations.

Once a thread blocks, the runtime engine selects the next thread to run. This is called *scheduling*. Threads shall be scheduled to run as long as they are not blocked. When all threads are blocked, simulated time is advanced according to the mode of execution in effect (see 1.4.3).

This standard does not address the scheduling problem. It is fully expected for thread scheduling to vary among compliant implementations for considerations such as performance. Well-formed ***e*** programs should be robust in the face of scheduling variations: the computation should yield compatible results irrespective of scheduling. ***e*** has several synchronization mechanisms to support robust concurrent programming, see Clause 11 and Clause 32. Well-formed ***e*** programs shall have congruent execution between software implementations based on simulated concurrency and hardware implementations, where concurrency is actual.

1.4.5.6 Functional coverage

Functional coverage is a user-defined metric. The process of measuring functional coverage involves three steps, as follows:

a) The programmer builds a coverage model, representing key architectural and micro-architectural features of the system being verified.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

b) Coverage data is collected during simulation and accumulated for analysis.

c) Coverage information is aggregated across many simulations and is analyzed to produce coverage scores. Coverage scores indicate the extent to which the features represented in the coverage models have been exercised by the verification process.

A coverage model includes one or more *cover groups*, which represent a data set to be sampled under certain conditions: the occurrence of a *sampling event* with an optional combinational guard. A cover group consists of *cover items*. Each cover item samples one value for each occurrence of the sampling event and classifies that value according to programmer-defined categories called *buckets*. During runtime, coverage values are sampled each time a sampling event is emitted. The trail of cumulative samples is the raw data used to determine coverage scores.

The creation of a coverage model entails the definition of cover groups with their sampling conditions, cover items, and buckets. Coverage model definition is intertwined into the rest of an ***e*** program, as coverage groups are added to various object types. Coverage is often added as a separate aspect, maintaining a clean separation from other functions.

Functional coverage is distinct from other coverage measures, such as code coverage, state machine coverage, assertion coverage, and the like. However, all these forms of coverage can be combined to form a multi-dimensional hybrid coverage metric. More information about functional coverage can be found in Clause 15 and in *Functional Verification Coverage Measurement and Analysis* [B5].[4]

1.4.5.7 Checking, printing, and reporting

e features several constructs aimed at simplifying the control over checking, printing, and reporting. These features might seem mundane, but there is great practical value in a common control scheme to which all ***e*** programs adhere, especially when combining components developed independently.

e programs can contain code to identify, and react to, various error conditions. One source of errors is unexpected behavior of the system being verified. These errors are called *DUT errors* (DUT stands for *design under test*). A second source of errors is assertions indicating unexpected behavior of the ***e*** program itself. A third kind of errors is exceptions created by the ***e*** runtime engine in response to unhandled programming errors.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

Each DUT error can be selectively assigned a degree of severity, from logging a message to immediate termination of the run. A customized error message that aids debugging can be associated with each DUT error. A DUT error can trigger the execution of a code block for logging and recovery. All of these features provide structure necessary for managing large verification environments containing massive amounts of unfamiliar code.

The checking, printing, and reporting constructs in ***e*** are discussed in Clause 17.

1.5 Conventions used

This standard uses visual cues to help locate and interpret information easily.

1.5.1 Visual cues (meta-syntax)

The meta-syntax for the description of lexical and syntax rules uses the conventions shown in Ta ble 2.

1.5.2 Syntax notation within a construct

Each construct subclause starts with the syntax for the construct. The syntax shows the construct, any arguments it accepts with their types, and the construct's return type (if it has one). The argument types and the construct return type are for information only and are not entered. When using the construct, follow the meta-syntax detailed in Ta ble 2.

Examples

The syntax notation for the predefined pseudo-method **first()** is

list.**first(***exp*: bool**)**: list-type

[4]The numbers in brackets correspond to those of the bibliography in Annex A.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

Table 2—Document conventions

Visual cue	Represents
courier	The courierfont indicates ***e*** or HDL code. For example, the following line indicates ***e*** code: keep opcode in [ADD, ADDI];
bold	The **bold** font is used to indicate keywords, text that shall be typed exactly as it appears. See also 1.6. For example, in the following command, the keywords "keep" and "reset_soft" as well as the period and the parentheses shall be typed as they appear: **keep** *item***.reset_soft()**
italic	The *italic* font represents user-defined variables. For example, a Boolean expression needs to be specified in the following line (after the "keep" keyword): **keep** *constraint-bool-exp*
[ ] square brackets	Square brackets indicate optional parameters. For example, in the following construct, the keywords "list of" are optional: **var** *name*: [**list of**] *type*
[] bold brackets	Bold square brackets are required. For example, in the following construct, the bold square brackets need to be typed as they appear: **extend enum-type-name**: **[***name***, ...]**

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

construct, ... *construct*; ...	An item followed by a separator character (usually a comma or a semicolon) and an ellipsis is an abbreviation for zero or more elements of the specified type, each separated from the next by the separator character. For example, in the following line zero or more actions may appear between the braces, each separated from the next by a semicolon. **if** *bool-exp* [**then**] {*action*; ...}
( ) parentheses	Parentheses [()] indicates a grouping, usually of alternative choices. For example, the following line shows the "bits" or "bytes" keywords are possible values for the "type" parameter: **type** *scalar-type* (**bits** | **bytes**: *num*)
| separator bar	The separator bar (|) character indicates alternative choices. For example, the following line indicates either the "bits" or "bytes" keyword shall be used: **type** *scalar-type* (**bits** | **bytes**: *num*)

~~*list*.**first(***exp*: bool): list-type~~

This is what the notation means~~:~~:

— The bold.**first** and the parentheses [**()**] shall be entered exactly.

— The parts in italics, *list* and *exp*, need to be replaced by a list name and an expression.

— ": bool" indicates the *exp* needs to be a Boolean expression.

— ": list-type" means the pseudo-method returns an item of the list element type.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

Following is an example of a call to the *list*.**first()** pseudo-method, where numbers is a list of integer items and my_number is an integer. The pseudo-method returns the first integer in the list greater than 5:

```
my_number = numbers.first(it > 5)
```

1.5.3 Syntax examples

Any syntax examples shown in this standard are for information only and are only intended to illustrate the use of such syntax.

1.6 Use of color in this standard

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in black and white. The places where color is used are the following:

— Cross references that are hyperlinked to other portions of this standard are shown in underlined-blue text (hyperlinking works when this standard is viewed interactively as a PDF file).

— Syntactic keywords and tokens in the formal language definitions are shown in **boldface-red text**.

1.7 Contents of this standard

The organization of the remainder of this standard is as follows:

— Clause 2 provides references to other applicable standards that are assumed or required for this standard.

— Clause 3 defines terms, acronyms, and abbreviations used throughout the different specifications contained in this standard.

— Clause 4 describes the fundamental syntactic and semantic components of the ***e*** language.

— Clause 5 defines ***e*** data types and describes their usage.

— Clause 6 defines how structs, subtypes, and fields function within this standard.

— Clause 7 describes the constructs used to define units and their use in a modular verification methodology.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

— Clause 8 describes the principles and usage of ***e*** templates features. Template types in ***e*** let the user define generic structs and units that are parameterized by type, similar to C++ templates. The user can later instantiate them, giving specific types as actual parameters.

— Clause 9 defines ***e*** unit interfaces and their usage within this standard.

— Clause 10 defines test generation and constraint functions within this standard.

— Clause 11 defines how temporal constructs can be used for specifying and verifying behavior over time in a ***e*** program.

— Clause 12 defines the syntax and semantics of TEs and describes their usage to track temporal behavior.

— Clause 13 defines two other struct members that can be used for temporal coding of events.

— Clause 14 defines what time-consuming actions are and how to use them.

— Clause 15 describes how to define, extend, and use coverage constructs.

— Clause 16 describes how to modify the grammar of the ***e*** language.

— Clause 17 shows the various constructs that print an expression, check for errors in the design under test (DUT), or add exception handling and diagnostics to an ***e*** program.

— Clause 18 defines how to declare and use ***e*** methods within an ***e*** program.

— Clause 19 describes how to create and assign values to ***e*** variables.

— Clause 20 shows how to perform basic packing and unpacking of scalars, strings, lists, and structs in this standard.

— Clause 21 describes how to use ***e*** control flow actions.

— Clause 22 defines how to create dependencies between ***e*** files and use the preprocessor directives in this standard.

— Clause 23 contains the syntax and descriptions of the ***e*** statements used to create packages and modify access control.

— Clause 24 describes simulation-related actions, expressions, and routines used.

— Clause 25 describes the flexible mechanism used to write text messages to various destinations.

— Clause 26 describes a uniform way to define streams of data items and compose them into verification scenarios.

— Clause 27 describes the pseudo-methods used to work with lists.

— Clause 28 describes all the predefined ~~routines~~methods used.

— Clause 29 describes all the predefined routines used.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

— Clause 30 contains information about using files and the predefined file routines.

— Clause 31 explains how to access structural information about the type system through the application programming interface (API), also known as the *reflection API,* and defines the reflection API.

— Clause 32 defines some predefined methods that are useful in controlling time-consuming methods (TCMs) and resource sharing between TCMs.

— Clause 33 describes IP protection, which aims to shield valuable information contained in an IP without sacrificing the usability of the IP as a whole.

— Annexes. Following Clause 33 are a series of annexes.

2. Normative references

The following referenced documents are indispensable for the application of this standard (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEC/IEEE 61691-1-1, Behavioral languages—Part 1-1: VHDL language reference manual.[5, 6]

IEEE Std 754™, IEEE Standard for Floating-Point Arithmetic.[7]

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

ISO/IEC 8859-1, Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin Alphabet No. 1.[8]

ISO/IEC 9899, Programming languages—C.

ISO/IEC/IEEE 9945, Information technology—Portable Operating System Interface (POSIX ~~UNIX Specification, Version 3~~®) Base- Specifications, Issue 7.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. *The ~~Authoritative Dictionary of~~ IEEE Standards Dictionary: Glossary of Terms & Definitions* should be ~~referenced~~consulted for terms not defined in this clause.[9]

3.1 Definitions

aspect: One or more program modules that contain refinements to existing types, typically to implement a specific feature.

attribute: A name, or name-value pair, associated with an object, typically used to classify or control object behavior.

bucket: A range of values that are collapsed together (are not distinguished) for the purpose of coverage measurement.

casting: The operation of changing the type associated with a value.

configuration: A data structure containing flags that control the operation of the runtime environment.

[5]IEC publications are available from the ~~Sales Department~~Central Office of the International Electrotechnical Commission, ~~Case Postale 131,~~ 3, rue de Varembé, P.O. Box 131, CH-1211, Geneva 20, Switzerland~~/Suisse~~ (http://www.iec.ch/). IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

[6]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854- 4141, USA (http://standards.ieee.org).

[7]The IEEE standards or products referred to in Clause 2 are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

[8]ISO publications are available from the ISO Central Secretariat, 1, ch. de la Voie-Creuse, Case Postale 56, ~~1 rue de Varembé,~~ CH-1211, Geneva 20, Switzerland~~/Suisse~~ (http://www.iso.org/). IEC publications are available from the Central Office of the International Electrotechnical Commission, 3, rue de Varembé, P.O. Box 131, CH-1211, Geneva 20, Switzerland (http://www.iec.ch/). ISO/IEC publications are also available in the United States from ~~Global Engineering Documents,~~

~~15 Inverness Way East, Englewood, CO 80112, USA (http://global.ihs.com/). Electronic copies are available in~~ the ~~United States from the~~Sales Department, American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

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

constraint: A language **construct** imposing some (optionally conditional) restrictions on the set of values that can be assigned to an object or a scalar.

construct: A component of the language.

coverage (functional)**:** A user-defined metric for measuring the thoroughness of functional verification.

coverage group: A set of coverage values that are sampled together each time an associated event is emitted.

event: A keyword used to define a signaling entity. Events can be either present or absent during each tick.

extension: A refinement of a **construct** in an **aspect**.

field: A feature of **struct**, used to hold values or references to objects.

generation: A process of assigning pseudo-random values to object hierarchies, according to applicable **constraints**.

global: A keyword referring to the top object in the runtime system.

inheritance: A property of a programming language, allowing a type to be defined as an extension to a previously defined type. The newly defined type is said to inherit from the previously defined type.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

keep: A keyword used to define **constraints**.

keyed list: A **list** containing members with **fields** designated as keys. Retrieval of list members using their keys is implied to be efficient, typically utilizing hashing techniques.

list: A telescopic data structure used to hold ordered collections of objects of specific types.

macro: A **construct** used to define a syntactic extension to the language and an associated semantics, in terms of the previously defined language.

method: A programming **construct** comprising a sequence of actions within an object context.

packing: The operation of marshaling a data structure, such as an object hierarchy, to a sequence of bits.

path (expression)**:** An expression composed of names or expressions returning names concatenated with a dot (.). A path expression allows reference to objects that are not in the immediate scope.

port: An interface feature of a **unit**.

pseudo-method: A **construct** that has a syntactic form of a **method**.

pseudo-routine: A **construct** that has a syntactic form of a routine.

sampling event: An **event** associated with a **temporal expression** (TE) that determines when the TE is evaluated.

state machine: A programming style for implementing synchronous automation.

struct: A keyword used to define a new object type.

subtype: A type inheriting from some other type.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

temporal expression (TE): An expression describing the behavior of a system in time, the system comprising a set of variables.

tick: An instance in time. The execution of a runtime environment is comprised of a sequence of discrete execution steps, one for each tick.

time-consuming method (TCM): A **method** containing actions that can block during execution. TCMs have internal state that persists during execution.

unit: A keyword used to define a type that is optionally associated with a structural component (an HDL module).

3.2 Acronyms and abbreviations

AO aspect-oriented

AOP aspect-oriented programming

API application programming interface

CRC cyclic redundancy check

CSP constraint satisfaction problem

DAG directed acyclic graph

DFS depth-first search

DUT design under test

EDA electronic design automation

ESI external simulator interface

FIFO first-in-first-out

HDL hardware description language

HEC header error control

IC integrated circuit iff if and only if

I/O input/output

IP intellectual property

LHO left-hand operand

LHS left-hand side

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

LSB least significant bit

MSB most significant bit

MVL multi-value logic

OO object-oriented

OS operating system

RHO right-hand operand

RHS right-hand side

SCC strongly connected component

TCM time-consuming method

TE temporal expression

TLM transaction-level modeling

VHDL VHSIC hardware description language (see IEC/IEEE 61691-1-1)

VHSIC very high-speed integrated circuit

4. *e* basics

This clause describes the structure of an ***e*** program, starting with the organization of ***e*** code into one or more files and the four categories of ***e*** constructs, and ending with a description of the *struct hierarchy*. This clause also describes the ***e*** operators.

4.1 Lexical conventions

The following subclauses describe the lexical conventions of ***e***. This standard uses the ASCII character set (see ISO/IEC 8859-1).

4.1.1 File structure

e code can be organized in multiple files. File names shall be legal ***e*** names. The default file extension is .e. ***e*** code files are sometimes referred to as *modules*. Each module contains at least one code segment and can also contain comments.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

4.1.2 Code segments

A *code segment* is enclosed with a begin-code marker **<'** and an end-code marker **'>**. Both the begin-code and the end-code markers shall be placed at the beginning of a line (left-most position), with no other text on that same line. For example, the following three lines of code form a code segment:

```
<'
import cpu_test_env
'>
```

Several code segments can appear in one file. Each code segment consists of one or more statements.

4.1.3 Comments and white space

e files begin as a *comment,* which ends when the first begin-code marker **<'** is encountered.

Comments within code segments can be marked with double dashes (**--**) or double slashes (//), terminated by an end-of-line character [a ~~carriage~~ return (enter), line-feed, or any combination of the two].

```
a = 5; -- This is an inline comment
b = 7 // This is also an inline comment
```

The end-code **'>** and the begin-code **<'** markers can be used in the middle of code sections, to write several consecutive lines of comment.

Example

Import the basic test environment for the CPU...

```
<'
import cpu_test_env
'>
```

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

This particular test requires the code that bypasses bug#72 as well as the constraints that focus on the immediate instructions.

```
<'
import bypass_bug72;
import cpu_test0012
'>
```

White space is one or more consecutive white space characters, including a space, tab, ~~carriage~~ return (enter), or line-feed.

4.1.4 Literals and constants

Literals are numeric, character, and string values specified literally in ***e***. Operators can be applied to literals to create compound expressions. The following categories of literals and constants are supported in ***e***:

— Unsized numbers

— Sized numbers

— Multi-value logic (MVL) literals

— Predefined constants

— Literal string

— Literal character

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

4.1.4.1 Unsized numbers

Unsized numbers are always positive and zero-extended, unless preceded by a hyphen (-). Decimal constants are treated as signed integers and have a default size of 32 bits. Binary, hex, and octal constants are treated as unsigned integers, unless preceded by a hyphen (-) to indicate a negative number, and have a default size of 32 bits. If the number cannot be represented in 32 bits, then it is represented as an unbounded integer (see 5.1.3).

The notations shown in Ta ble 3 can be used to represent unsized numbers.

Table 3—Representing unsized numbers in expressions

Notatio	Legal	Examples
Decimal integer	Any combination of 0–9, possibly preceded by a hyphen (-) for negative numbers. An	12 55_32
Binary integer	Any combination of 0–1, preceded by **0b**. An underscore (_) can be added anywhere in the	0b100111 0b1100_0101
Hexadecimal integer	Any combination of 0–9and a–f, preceded by **0x**. An underscore (_) can be added anywhere	0xff 0x99_aa_bb_cc
Octal integer	Any combination of 0–7, preceded by **0o**. An underscore (_) can be added anywhere in the	0o66_123
K (multiply by 1024)	A decimal integer followed by a **K** or **k**. For example,	32K 32k
M (multiply by 1024*1024)	A decimal integer followed by an **M** or **m**. For example, 2m = 2 097 152.	1m 4m 4M

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

4.1.4.2 Sized numbers

A *sized number* is a notation that defines a literal with a specific size in bits. The syntax is:

width-number ' (**b** | **o** | **d** | **h** | **x**) *value-number*

where *width-number* is a decimal integer specifying the width of the literal in bits, and *value-number* is the value of the literal, specified as one of four radixes, as shown in Ta ble 4.

Table 4—Radix specification characters

Radix	Represented by	Example
Binary	A leading '**b** or '**B**. An underscore (_) can be added anywhere in the number for readability.	8'b1100_1010
Octal	A leading '**o** or '**O**. An underscore (_) can be added anywhere in the number for readability.	6'o45
Decimal	A leading '**d** or '**D**. An underscore (_) can be added anywhere in the number for readability.	16'd63453
Hexadecimal	A leading '**h** or '**H** or '**x** or '**X**. An underscore (_) can be added anywhere in the number for readability.	32'h12ff_ab04

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

If *value-number* is more than the specified size in bits, its most significant bits (MSBs) are ignored. If *value-number* is less that the specified size, it is padded on the left with zeros (0).

Sized numbers cannot contain Mor Kmultipliers.

4.1.4.3 MVL literals

An *MVL literal* is based on the mvl type, which is a predefined enumerated scalar type in ***e***. The *mvl type* is defined as:

```
type mvl : [MVL_U,MVL_X,MVL_0,MVL_1,MVL_Z,MVL_W,MVL_L,MVL_H,MVL_N]
```

NOTE—MVL_Nrepresents "don't care." [10]

If a port is defined as type list of mvl, values can be assigned by using the **$** access operator, e.g.,

```
sig$ = {MVL_X; MVL_X; MVL_X}
```

If the port is a numeric type (uint, int, and so on), mvl values can be assigned by using the predefined MVL methods for ports, e.g.,

```
sig.put_mvl_list({MVL_X; MVL_X; MVL_X})
```

An MVL literal, which is a literal of type list of mvl, provides a more convenient syntax for assigning MVL values. The syntax of an MVL literal is:

width-number ' (**b** | **o** | **h**) *value-number*

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

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

where *width-number* is an unsigned decimal integer specifying the size of the list and *value-number* is any sequence of digits that are legal for the base, plus **x**, **z**, **u**, **l**, **h**, **w**, and **n**, as shown in Table 5.

A *value-number* can not be defined by using a decimal base.

Table 5—MVL values

Value	Definitio
U	Uninitialized
X	Forcing unknown
0	Forcing 0
1	Forcing 1
Z	High impedance
W	Weak unknown
L	Weak 0
H	Weak 1
N	Don't care

4.1.4.3.1 Syntax rules

a) A single digit represents four bits in hexadecimal base, three bits in octal base, and one bit in binary base. Similarly, the letters **x**, **z**, **u**, **l**, **h**, **w**, and **n** represent four identical bits (for hexadecimal), three identical bits (for octal), or one bit (for binary). For example, 8'h1x is equivalent to 8'b0001xxxx.

b) If the size of the value is smaller than the width, the value is padded to the left. An MSB of 0or 1 causes zero-padding (0). If the MSB of the literal is **x**, **z**, **u**, **l**, **h**, **w**, or **n**, that mvl value is used for padding.

c) If the size of the value is larger than the size specified for the list, the value is truncated, leaving the least significant bit (LSB) of the literal.

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008

d) An underscore can be used for breaking up long numbers to enhance readability. It is legal inside the size and inside the value. It is not legal at the beginning of the literal, between the size and the single quote ('), between the base and the value, and between the single quote (') and the base.

e) Decimal literals are not supported.

f) White space shall not be used as a separator between the width number and base or between the base and the value.

g) The base and the value are not case-sensitive. h) Size

and base values need to be specified.

i) In the context of a Verilog comparison operator (**!==** or **===**) or HDL tick access (e.g., 'data' = 32'bx), only the 4-value subset is supported (**0**, **1**, **u**, or **x**).

j) Verilog simulators support only the 4-value logic subset.

k) An MVL literal of size 1is of type list of mvl that has one element. It is not of type mvl. Thus, an MVL literal cannot be assigned to a variable or field of type mvl.

l) The type-casting operations **as_a()** and **is a** do not propagate the context.

m) If the type of the expression is numeric (based on its context) or if the type cannot be extracted from the context, the default type remains uint.

n) Syntactically, the same expression can be a numeric type or MVL literal. For example, 1'b1 can represent the number one (1) or a list of MVL with the value {MVL_1}.

4.1.4.3.2 Examples

32'hffffxxxx

32'HFFFFXXXX

//16'_b1100uuuuu --illegal because (_) is between (') and base

19'oL0001

14'D123 -- illegal because decimal literals are not supported

64'bz_1111_0000_1111_0000

5.3876e4

1e+5

7.321E-3

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

4.1.4.3.3 Considerations

A literal is considered to be an MVL literal when it is:

— assigned to a list of mvl, e.g., var v2: list of mvl = 16'b1;

— passed to a method that receives a list of mvl

— assigned to a port of type list of mvl using the **$** operator

— compared to list of mvl, e.g., check that v == 4'buuuu;

— compared using the **===** and **!==** operators, e.g., check that 's' === 4'bz;

— used in an HDL tick access assignment, e.g., 's' = 8'bx1z;

— an argument for a Verilog task, e.g., 'task1'(8'h1x)

— used in a list operation, e.g., var l: list of mvl; l.add(32'b0)

In those contexts where both numeral literals and MVL literals can be accepted, a numeric literal shall be inferred, unless the literal contains the characters x, z, u, l, h, w, and n.

4.1.4.4 Predefined constants

The set of ***e*** *predefined constants* is shown in Ta ble 6.

4.1.4.5 Literal string

A *literal string* is a sequence of zero or more ASCII characters enclosed by double quotes (" "). The escape sequences shown in Ta ble 7 can also be used to specify special characters, e.g., a tab (\t).

Any combination of escape characters that is not listed in Ta ble 7 have no special meaning. In this case, the backslash (\) character (and all the preceding characters) appear in the literal string.

Documents
Document Sharing Hub

Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.

Full document details: IEEE 1647-2008

Table 6—Predefined constants

Constant	Descriptio
TRUE	Used for Boolean variables and expressions.
FALSE	Used for Boolean variables and expressions.
NULL	Used for structs, this specifies a NULLpointer; within character strings, this
UNDEF	UNDEFsignifies NONEwhere an index is expected. UNDEFhas the value -1.
MAX_INT	Represents the largest 32-bit **int** (2^{31}–1)
MIN_INT	Represents the smallest 32-bit **int** (–2^{31}).
MAX_UINT	Represents the largest 32-bit **uint** (2^{32}–1).

Table 7—Escape sequences in strings

Escape sequence	Meanin
\n	Newline
\t	Tab
\f	Form-feed
\"	Quote
\\	Backslash
\r	~~Carriage return~~Return

4.1.4.6 Literal character

A *literal character* is a single ASCII character, enclosed in quotation marks and preceded by **0c**. This expression evaluates to the integer value that represents this character. For example, the following literal character is the single ASCII character a and evaluates to 0x0061:

```
var u: uint(bytes:2) = 0c"a"
```

Documents
Document Sharing Hub
Authorized licensed use limited to: Tsinghua University. Downloaded on February 24,2026 at 21:40:57 UTC from IEEE Xplore. Restrictions apply.
Full document details: IEEE 1647-2008