



IEEE Standard for the Functional Verification Language 'e'

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

1647™

IEEE
3 Park Avenue
New York, NY 10016-5997, USA
29 September 2006

IEEE Std 1647™-2006

*Recognized as an
American National Standard (ANSI)*

IEEE Std 1647™-2006

IEEE Standard for the Functional Verification Language 'e'

Sponsor

**Design Automation Standards Committee
of the
IEEE Computer Society**

Approved 8 August 2006

American National Standards Institute

Approved 30 March 2006

IEEE-SA Standards Board

Abstract: The *e functional verification language* is an application-specific programming language, aimed at automating the task of verifying an electronic 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. This standard contains a definition of the *e* language syntax and semantics, and how tool developers and verification engineers should use them.

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

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

Copyright © 2006 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 29 September 2006. Printed in the United States of America.

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

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

PRINT: ISBN 0-7381-4940-3 SH95524
PDF: ISBN 0-7381-4941-1 SS95524

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.

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 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. 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 IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

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

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

Introduction

This introduction is not part of IEEE Std 1647-2006, 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 electronic 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, ST Microelectronics, 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 programming model, with implicit memory management and single inheritance. In this, *e* is similar to Java.
- 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 like Verilog[®] (see IEEE Std 1364[™])^a and VHDL (see IEC/IEEE 61691-1-1). Concurrency in *e* is synchronous, like in Esterel.
- *e* contains a temporal language that borrows from Linear Temporal Logic and Interval Temporal Logic.
- *e* has many built-in constructs aimed at simplifying common programming tasks; built-in support for lists and hashes, which take after Eiffel[™]; and pattern matching and string- and file-manipulation features, which are borrowed from Perl[™].

^aInformation on references can be found in Clause 2.

- 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

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.

Participants

The following members and observers took part in the *e* Functional Verification Language Working Group:

Yaron Kashai, *Chair*
Joe Daniels, *Technical Editor*

Alan Ahlschlager	Olivier Haller	Daniel P. Romaine
Cristian Amitroaie	Joseph V. Hupcey III	Sharon Rosenberg
Christoffer Amlo	Sylvie Hurat	Ken Sailor
Maarten Arts	Achutha Jois	Jun Satoh
Victor Berman	Eran Keydar	Andrey Schwartz
Avi Bloch	Adiel A. Khan	Eric Sherk
Ronald A. Bow	Zeev Kirshenbaum	Tim Short
Levent Caglar	Pete LaFauci	Uwe Simm
Serrie-Justine Chapman	Vitaly Lagoon	Lori Kate Smith
Anand V. Chavan	Brett G. Lammers	Javier Sosa
Kamaraj Chermaswamy	Frank G. Lang	Alexandru G. Stanculescu
Ric P. Chope	Chris Macionski	Michael K. Stellfox
Sarah Lynne Cooper	David Malandain	Mark D. Strickland
Sameer Dani	Silas McDermott	Rajesh Thyagarajan
Jacob Daniel	Michael T. McNamara	Dave Tokic
Sandeep S. Desai	Richard Mitchell	Sriram Tyagarajan
Iraklis I. Diamantidis	Matthew J. Morley	David Van Campenhout
Stylianios Diamantidis	Bryan J. Morris	Matan Vax
Dean R. D'Mello	John C. Morris	David G. Von Bank
Darren Galpin	Meirav Nitzan	Henry J. Von Bank
Ann Germany	Patrick Oury	Yaron Wolfsthal
John C. Goss	Meir Ovadia	Naomi M. Wordsworth
Pitchumani Guruswamy	Steven M. Pearlmutter	Frazer J. Worley
Geoffrey Hall	Stewart Penman	Vitaly Yankelevich
Giles T. Hall	Andrew J. Piziali	Amos Ziv
	Nilesh D. Ranpura	

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

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

When the IEEE-SA Standards Board approved this standard on 30 March 2006, it had the following membership:

Steve M. Mills, *Chair*
Richard H. Hulett, *Vice Chair*
Don Wright, *Past Chair*
Judith Gorman, *Secretary*

Mark D. Bowman
Dennis B. Brophy
William R. Goldbach
Arnold M. Greenspan
Robert M. Grow
Joanna N. Guenin
Julian Forster*
Mark S. Halpin
Kenneth S. Hanus

William B. Hopf
Joseph L. Koepfinger*
David J. Law
Daleep C. Mohla
T. W. Olsen
Glenn Parsons
Ronald C. Petersen
Tom A. Prevost

Greg Ratta
Robby Robson
Anne-Marie Sahazizian
Virginia C. Sulzberger
Malcolm V. Thaden
Richard L. Townsend
Walter Weigel
Howard L. Wolfman

*Member Emeritus

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

Satish K. Aggarwal, *NRC Representative*
Richard DeBlasio, *DOE Representative*
Alan H. Cookson, *NIST Representative*

Michael D. Fisher
IEEE Standards Program Manager, Document Development

Mike Kipness
IEEE Standards Program Manager, Technical Program Development

Contents

1.	Overview.....	1
1.1	Scope.....	1
1.2	Purpose.....	1
1.3	Basic concepts relating to this standard.....	2
1.4	Conventions used.....	8
1.5	Contents of this standard.....	9
2.	Normative references.....	11
3.	Definitions, acronyms, and abbreviations.....	11
3.1	Definitions.....	11
3.2	Acronyms and abbreviations.....	13
4.	<i>e</i> basics.....	15
4.1	Lexical conventions.....	15
4.2	Syntactic elements.....	22
4.3	Struct hierarchy and name resolution.....	28
4.4	Ranges.....	34
4.5	Operator precedence.....	35
4.6	Evaluation order of expressions.....	36
4.7	Bitwise operators.....	36
4.8	Boolean operators.....	38
4.9	Arithmetic operators.....	40
4.10	Comparison operators.....	41
4.11	String matching.....	45
4.12	Extraction and concatenation operators.....	48
4.13	Scalar modifiers.....	52
4.14	Parentheses.....	52
4.15	list.method().....	53
4.16	Special-purpose operators.....	53
5.	Data types.....	57
5.1	<i>e</i> data types.....	57
5.2	Untyped expressions.....	62
5.3	Assignment rules.....	63
5.4	Precision rules for numeric operations.....	66
5.5	Automatic type casting.....	68
5.6	Defining and extending scalar types.....	68
5.7	Explicit type conversion.....	71
6.	Structs, subtypes, and fields.....	77
6.1	Structs overview.....	77
6.2	Defining structs: struct.....	77
6.3	Extending structs: extend type.....	78
6.4	Restrictions on inheritance.....	79
6.5	Extending subtypes.....	79

6.6	Creating subtypes with when	79
6.7	Extending when subtypes	81
6.8	Defining fields: field	82
6.9	Defining list fields.....	83
6.10	Projecting list of fields	85
6.11	Defining attribute fields	85
7.	Units.....	87
7.1	Overview	87
7.2	Defining units and fields of type unit	89
7.3	Unit attributes.....	93
7.4	Predefined methods of any_unit	94
7.5	Unit-related predefined methods of any_struct.....	96
7.6	Unit-related predefined routines	98
8.	<i>e</i> ports.....	101
8.1	Introduction to <i>e</i> ports.....	101
8.2	Using simple ports	102
8.3	Using buffer ports	104
8.4	Using event ports	105
8.5	Defining and referencing ports	106
8.6	Port attributes	109
8.7	Buffer port methods	121
8.8	MVL methods for simple ports.....	123
8.9	Global MVL routines.....	129
8.10	Comparative analysis of ports and tick access.....	132
9.	Constraints and generation.....	135
9.1	Types of constraints	135
9.2	Generation concepts.....	135
9.3	Defining constraints	148
9.4	Invoking generation	155
10.	Events.....	157
10.1	Causes of events.....	157
10.2	Scope of events	157
10.3	Defining and emitting named events	157
10.4	Predefined events	158
11.	Temporal expressions	161
11.1	Overview	161
11.2	Temporal operators and constructs	164
11.3	Success and failure of a temporal expression	179
12.	Temporal struct members	181
12.1	on	181
12.2	expect assume	182

13.	Time-consuming actions.....	183
	13.1 Synchronization actions.....	183
	13.2 Concurrency actions.....	184
	13.3 State machines	186
14.	Coverage constructs.....	191
	14.1 Defining coverage groups: cover	191
	14.2 Defining basic coverage items: item	193
	14.3 Defining cross coverage items: cross	197
	14.4 Defining transition coverage items: transition	199
	14.5 Extending coverage groups: cover ... using also ... is also	201
	14.6 Extending coverage items: item ... using also	202
	14.7 Coverage API methods	202
	14.8 Coverage methods for the covers struct.....	206
15.	Macros.....	211
	15.1 Syntax overview.....	211
	15.2 Defining a macro.....	211
	15.3 Match expression structure	212
	15.4 Interpretation of match expressions	213
	15.5 Replacement code	214
16.	Print, checks, and error handling	217
	16.1 print	217
	16.2 Handling DUT errors	217
	16.3 Handling user errors.....	221
	16.4 Handling programming errors: assert	223
17.	Methods.....	225
	17.1 Rules for defining and extending methods	225
	17.2 Invoking methods.....	233
	17.3 Parameter passing	236
	17.4 Using the C interface	238
18.	Creating and modifying <i>e</i> variables	241
	18.1 About <i>e</i> variables	241
	18.2 var	241
	18.3 = ...	242
	18.4 op=	242
	18.5 <=	243
19.	Packing and unpacking	245
	19.1 Basic packing	245
	19.2 Predefined pack options.....	248
	19.3 Customizing pack options.....	249
	19.4 Packing and unpacking specific types	249
	19.5 Implicit packing and unpacking.....	254

20.	Control flow actions.....	257
	20.1 Conditional actions	257
	20.2 Iterative actions.....	259
	20.3 File iteration actions.....	263
	20.4 Actions for controlling the program flow	264
21.	Importing and preprocessor directives.....	265
	21.1 Importing <i>e</i> modules	265
	21.2 <code>#ifdef</code> , <code>#ifndef</code>	266
	21.3 <code>#define</code>	267
	21.4 <code>#undef</code>	268
22.	Encapsulation constructs.....	269
	22.1 package: package-name	269
	22.2 package: type-declaration	269
	22.3 package protected private: struct-member	270
23.	Simulation-related constructs.....	271
	23.1 force	271
	23.2 release	271
	23.3 Tick access: 'hdl-pathname'	272
	23.4 simulator_command().....	272
	23.5 stop_run()	273
24.	List pseudo-methods library	275
	24.1 Pseudo-methods overview	275
	24.2 Using list pseudo-methods.....	275
	24.3 Pseudo-methods to modify lists.....	275
	24.4 General list pseudo-methods.....	284
	24.5 Math and logic pseudo-methods	298
	24.6 List CRC pseudo-methods	300
	24.7 Keyed list pseudo-methods.....	302
25.	Predefined methods library	305
	25.1 Predefined methods of sys	305
	25.2 Predefined methods of any_struct.....	305
	25.3 Methods and predefined attributes of unit any_unit	308
	25.4 Pseudo-methods	308
	25.5 Coverage methods.....	310
26.	Predefined routines library.....	311
	26.1 Deep copy and compare routines	311
	26.2 Arithmetic routines	314
	26.3 bitwise_op()	318
	26.4 get_all_units()	319
	26.5 String routines.....	319
	26.6 Output routines.....	327

26.7	Operating system interface routines.....	329
26.8	set_config()	332
27.	Predefined file routines library	333
27.1	File names and search paths.....	333
27.2	File handles	333
27.3	Low-level file methods	333
27.4	General file routines.....	338
27.5	Reading and writing structs	344
28.	Predefined resource sharing control structs	349
28.1	Semaphore methods	349
28.2	How to use the semaphore struct	350
Annex A	(normative) Source code serialization.....	355
Annex B	(informative) Comparison of when and like inheritance.....	363
Annex C	(informative) Bibliography.....	371

IEEE Standard for the Functional Verification Language 'e'

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 standard is the definition of the *e* functional verification language. The *e* functional verification language is an application-specific programming language, aimed at automating the task of verifying electronic designs with respect to their specifications.

This standard aims to serve as an authoritative source for the definition of the following:

- a) syntax and semantics of *e* language constructs
- b) the *e* language interaction with standard simulation languages
- c) *e* language libraries

1.2 Purpose

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.

- 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 the output matches the expectations, based on the functional requirements, the state of the system being verified, and the inputs provided.
- 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 purpose of the *e* functional verification language is to facilitate the creation of sophisticated verification environments. *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.

1.3 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*.

1.3.1 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.

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

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*.

1.3.2 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 21 and Annex A, 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.

1.3.3 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 10 and Clause 11.

1.3.3.1 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 23.5, 26.8, and Clause 16.

1.3.3.2 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.

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 10.4.2.

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).

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 8 and Clause 23.

1.3.4 Execution phases

The execution of *e* programs goes through the phases listed in Table 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 TCMs in the context of an object; execute the timed portion of the program.	run()
extract	Collect log and coverage information after execution.	sys.extract()
check	Perform final checking.	sys.check()
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**.

For more information about the object instance hierarchy in an *e* program, see 4.3.1. The methods defined **any_struct** are discussed in 25.2.

1.3.5 Major semantic components

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

1.3.5.1 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 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 B.

1.3.5.2 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 Y is visible within Package P. Z is 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 Q and Type B.

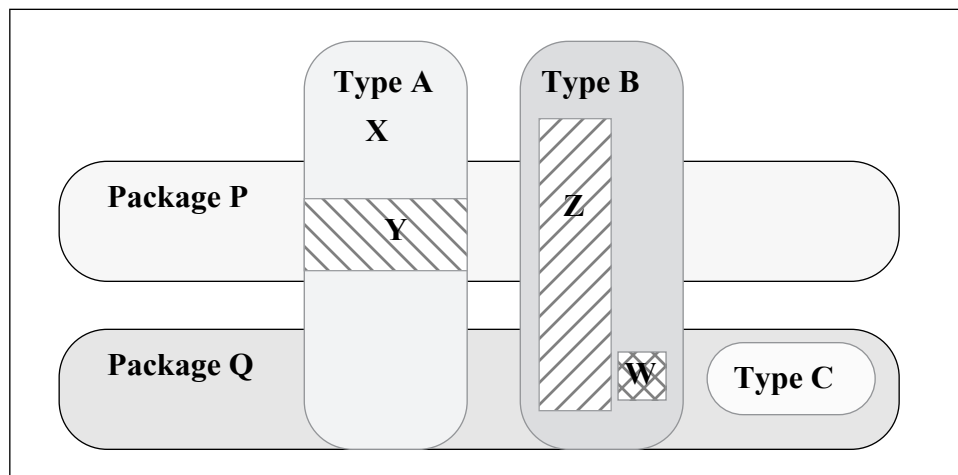


Figure 1—Type and package membership

For a detailed discussion of packages and information hiding, see Clause 22.

1.3.5.3 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).² Semantics of specific bit-oriented, logical, and arithmetic operators can be found in 4.7, 4.8, and 4.9, respectively.

²For information on references, see Clause 2.

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 8.8.

e does not support floating point numbers.

1.3.5.4 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 0 to 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.

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 9.

1.3.5.5 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 10 and Clause 13 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 1.3.3).

This standard does not address the scheduling problem. It is fully expected that thread scheduling will 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 10 and Clause 28. Well-formed *e* programs shall have congruent execution between software implementations based on simulated concurrency and hardware implementations, where concurrency is actual.

1.3.5.6 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.
- 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 14 and in *Functional Verification Coverage Measurement and Analysis* [B5].³

1.3.5.7 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 misnomer often used to describe the system being verified). A second source of errors are assertions indicating unexpected behavior of the *e* program itself. A third kind of errors are exceptions created by the *e* runtime engine in response to unhandled programming errors.

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 these features provide structure necessary for managing large verification environments containing massive amounts of unfamiliar code.

³The number in brackets correspond to those of the bibliography in Annex C.

The checking, printing, and reporting constructs in *e* are discussed in Clause 16.

1.4 Conventions used

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

1.4.1 Visual cues (meta-syntax)

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

Table 2—Document conventions

Visual cue	Represents
<i>courier</i>	The <i>courier</i> font indicates <i>e</i> or HDL code. For example, the following line indicates <i>e</i> code: <code>keep opcode in [ADD, ADDI];</code>
bold	The bold font is used to indicate keywords, text that shall be typed exactly as it appears. 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: <code>keep item.reset_soft()</code>
<i>italic</i>	The <i>italic</i> font represents user-defined variables. For example, a Boolean expression needs to be specified in the following line (after the “keep” keyword): <code>keep constraint-bool-exp</code>
[] square brackets	Square brackets indicate optional parameters. For example, in the following construct, the keywords “list of” are optional: <code>var name: [list of] type</code>
[] bold brackets	Bold square brackets are required. For example, in the following construct, the bold square brackets need to be typed as they appear: <code>extend enum-type-name: [name,...]</code>
<i>construct, ...</i>	An item, followed by a separator (usually a comma or a semicolon) and an ellipsis is an abbreviation for a list of elements of the specified type. For example, the following line means a list of zero or more names, separated by commas, can be used here: <code>extend enum-type-name: [name,...]</code>
	The separator bar () character indicates alternative syntax or parameters. For example, the following line indicates either the “bits” or “bytes” keyword shall be used: <code>type scalar-type (bits bytes: num)</code>

1.4.2 Syntax notation within a construct

Each construct section 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 Table 2.

Examples

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

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.

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 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 and acronyms 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.
- Clause 8 defines *e* unit interfaces and their usage within this standard.
- Clause 9 defines test generation and constraint functions within this standard.
- Clause 10 defines how temporal constructs can be used for specifying and verifying behavior over time in a *e* program.
- Clause 11 defines the syntax and semantics of TEs and describes their usage to track temporal behavior.
- Clause 12 defines two other struct members that can be used for temporal coding of events.
- Clause 13 defines what time-consuming actions are and how to use them.
- Clause 14 describes how to define, extend, and use coverage constructs.
- Clause 15 describes how to modify the grammar of the *e* language.
- Clause 16 shows the various constructs that print an expression, check for errors in the DUT, or add exception handling and diagnostics to an *e* program.
- Clause 17 defines how to declare and use *e* methods within an *e* program.
- Clause 18 describes how to create and assign values to *e* variables.
- Clause 19 shows how to perform basic packing and unpacking of scalars, strings, lists, and structs in this standard.
- Clause 20 describes how to use *e* control flow actions.
- Clause 21 defines how to create dependencies between *e* files and use the preprocessor directives in this standard.
- Clause 22 contains the syntax and descriptions of the *e* statements used to create packages and modify access control.
- Clause 23 describes simulation-related actions, expressions, and routines used.
- Clause 24 describes the pseudo-methods used to work with lists.

- Clause 25 describes all the predefined methods used.
- Clause 26 describes all the predefined routines used.
- Clause 27 contains information about using files and the predefined file routines.
- Clause 28 defines some predefined methods that are useful in controlling time-consuming methods (TCMs) and resource sharing between TCMs.
- Annexes. Following Clause 28 are a series of annexes.

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.

IEC/IEEE 61691-1-1, Behavioural languages—Part 1: VHDL language reference manual.^{4, 5}

IEEE Std 1364TM, IEEE Standard for Verilog Hardware Description Language.⁶

ISO/IEC 8859-1, Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin Alphabet No. 1.⁷

ISO/IEC 9899, Programming languages—C.

ISO/IEC 9945, POSIX UNIX Specification, Version 3.

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B3] should be referenced for terms not defined in this clause.

3.1 Definitions

3.1.1 aspect: A program module that contains refinements to existing types, typically to implement a specific feature.

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

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

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

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

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

3.1.7 construct: A component of the language.

⁴IEC publications are available from the Sales Department of the International Electrotechnical Commission, Case Postale 131, 3, rue de Varembe, 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, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (<http://www.ansi.org/>).

⁵IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (<http://standards.ieee.org/>).

⁶The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

⁷ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembe, CH-1211, Genève 20, Switzerland/Suisse (<http://www.iso.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 American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (<http://www.ansi.org/>).

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

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

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

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

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

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

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

3.1.15 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.

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

3.1.17 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.

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

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

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

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

3.1.22 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.

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

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

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

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

3.1.27 state machine: A programming style for implementing synchronous automation.

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

3.1.29 subtype: A type inheriting from some other type.

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

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

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

3.1.33 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
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
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 components
TCM	time-consuming method
TE	temporal expression
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.

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, 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;
>
```

```
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, 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

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 5.1.2.3).

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

Table 3—Representing unsized numbers in expressions

Notation	Legal characters	Examples
Decimal integer	Any combination of 0–9, possibly preceded by a hyphen (–) for negative numbers. An underscore (–) can be added anywhere in the number for readability.	12 55_32 –764
Binary integer	Any combination of 0–1, preceded by 0b . An underscore (–) can be added anywhere in the number for readability.	0b100111 0b1100_0101
Hexadecimal integer	Any combination of 0–9 and a–f, preceded by 0x . An underscore (–) can be added anywhere in the number for readability.	0xff 0x99_aa_bb_cc
Octal integer	Any combination of 0–7, preceded by 0o . An underscore (–) can be added anywhere in the number for readability.	0o66_123
K (multiply by 1024)	A decimal integer followed by a K or k . For example, 32K = 32768.	32K 32k 128k
M (multiply by 1024*1024)	A decimal integer followed by an M or m . For example, 2m = 2097152.	1m 4m 4M

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 radices, as shown in Table 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

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

Sized numbers cannot contain M or K multipliers.

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_N represents “don’t care.”⁸

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*

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

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.

Table 5—MVL values

Value	Definition
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

One cannot define *value-number* by using a decimal base.

4.1.4.3.1 Syntax rules

- 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.
- If the size of the value is smaller than the width, the value is padded to the left. An MSB of 0 or 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.
- 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.
- 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.
- Decimal literals are not supported.
- White space shall not be used as a separator between the width number and base or between the base and the value.
- The base and the value are not case-sensitive.
- Size and base values need to be specified.
- 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**).
- Verilog simulators support only the 4-value logic subset.
- An MVL literal of size 1 is 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.
- 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
```

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 Table 6.

Table 6—Predefined constants

Constant	Description
TRUE	Used for Boolean variables and expressions.
FALSE	Used for Boolean variables and expressions.
NULL	Used for structs, this specifies a NULL pointer; within character strings, this specifies an empty string.
UNDEF	UNDEF signifies NONE where an index is expected. UNDEF has 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 ($2^{32}-1$).

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 Table 7 can also be used to specify special characters, e.g., a tab (\t).

Table 7—Escape sequences in strings

Escape sequence	Meaning
\n	New-line
\t	Tab
\f	Form-feed
\"	Quote
\\	Backslash
\r	Carriage-return

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

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"
```

Without explicit casting, literal characters can only be assigned to integers or unsigned integers.

4.1.5 Names, keywords, and macros

The following subclauses describe the legal syntax for names and macros.

4.1.5.1 Legal *e* names

User-defined names in *e* code consist of a case-sensitive combination of any length, containing the characters A–Z, a–z, 0–9, and underscore (_). They shall begin with a letter. A field name, however, can begin with an underscore (_); this makes the field private to the module (the *e* file in which it appears).

4.1.5.2 *e* file names

An *e* module name (a file name) can contain characters only from the Portable Filename Character Set (see ISO/IEC 9945), except that only one dot (.) is allowed in a file name.

4.1.5.3 *e* keywords

The keywords listed in Table 8 are the components of the *e* language. Some of the terms are keywords only when used together with other terms, such as `key` in **list(key:key)** or `before` in **keep gen *x* before *y***.

Table 8—Keywords

all of	and	as_a	assert	assume
attribute	before	bit	bits	bool
break	byte	c export	case	change
check that	compute	computed	consume	continue
cover	cross	cycle	default	define
delay	detach	do	down to	each
edges	else	emit	event	exec
expect	extend	fail	fall	file
first of	for	force	from	gen
global	if	ifdef	ifndef	in
index	int	is	is a	is also
is c routine	is empty	is first	is inline	is instance
is not a	is not empty	is only	is undefined	item
keep	keeping	key	like	line
list of	matching	me	nand	new
nor	not	not in	now	on
only	or	pass	prev	print
range	ranges	release	repeat	return
reverse	rise	routine	select	soft
start	state machine	step	string	struct
sync	sys	that	then	time
to	transition	true	try	type
uint	unit	until	using	var
when	while	with		

4.1.5.4 Macros

e preprocessor names (declared by **#define** statements) consist of an identifier, possibly preceded by a backtick (`). A preprocessor name with backtick is also called a *Verilog-style define*. See 21.3.

4.1.5.5 String matching pseudo-variables

A successful string match results in assigning the local pseudo-variables \$1 to \$27 with the substrings corresponding to the non-blank meta-characters present in the pattern. For more details, see 4.11.

4.2 Syntactic elements

Every *e* construct belongs to a construct category that determines how the construct can be used. There are four categories of *e* constructs, as shown in Table 9.

Table 9—Construct categories

Category	Description
Statements	Statements are top-level constructs and are valid within the begin-code '<' and end-code '>' markers. See 4.2.1 for a list and brief description of <i>e</i> statements.
Struct members	Struct members are second-level constructs and are valid only within a struct definition. See 4.2.2 for a list and brief description of <i>e</i> struct members.
Actions	Actions are third-level constructs and are valid only when associated with a struct member, such as a method or an event. See 4.2.3 for a list and brief description of <i>e</i> actions.
Expressions	Expressions are lower-level constructs that can be used only within another <i>e</i> construct. See 4.2.4 for a list and brief description of <i>e</i> expressions.

The syntax hierarchy roughly corresponds to the level of indentation shown, as follows:

```
statements
  struct members
    actions
      expressions
```

4.2.1 Statements

Statements are the top-level syntactic constructs of the *e* language and perform the functions related to extending the *e* language and interface with the simulator. Statements are valid within the begin-code '<' and end-code '>' markers. They can extend over several lines and are separated by semicolons (;). For example, the following code segment has two statements:

```
<'
  import bypass_bug72;
  struct foo{};
'>
```

In general, within a given *e* module, statements can appear in any order, except **import** statements shall appear before any other statements, and only preprocessor directives or any defines (**#ifdef**, **#ifndef**, **define**, or **define as**) can precede **import** statements.

Table 10 shows the complete list of *e* statements.

Table 10—Statements

Statement	Description
struct	Defines a new data structure (see 6.2).
type	Defines an enumerated data type or scalar subtype (see 5.6.1, 5.6.2, or 5.6.3).
extend	Modifies a previously defined struct or type (see 5.6.4 or 6.3).
define-as	Extends the <i>e</i> language by defining new statements, struct members, actions, or expressions (see 15.2).
#define, #ifdef, #ifndef, #undef	Place conditions on the <i>e</i> parser (see Clause 21).
import	Declares dependency between two <i>e</i> modules (see 21.1.1).
unit	Defines a data struct associated with an HDL component or block (see 7.2.1).
C routine, C export	Allows C code to be called from <i>e</i> (see 17.4).

4.2.2 Struct members

Struct member declarations are second-level syntactic constructs of the *e* language that associate the entities of various kinds with the enclosing struct. Struct members can only appear inside a struct type definition statement (see 6.2). They can extend over several lines and are separated by semicolons (;). For example, the following struct `packet` has two struct members, `len` and `data`:

```
struct packet{
    %len: int;
    %data[len]: list of byte};
```

A struct can contain multiple struct members of any type in any order. Table 11 gives a brief description of each *e* struct member.

Table 11—Struct members

Declaration	Description
Field declaration	Defines a data entity that is a member of the enclosing struct and has an explicit data type.
Method declaration	Defines an operational procedure that can manipulate the fields of the enclosing struct and access runtime values in the DUT.
Subtype declaration	Defines an instance of the parent struct in which specific struct members have particular values or behavior.
Constraint declaration	Influences the distribution of values generated for data entities and the order in which values are generated.

Table 11—Struct members (continued)

Declaration	Description
Coverage declaration	Defines functional test goals and collects data on how well the testing is meeting those goals.
Temporal declaration	Defines <i>e</i> events and their associated actions.
#ifdef, #ifndef	Place conditions on the <i>e</i> parser (see 21.2).

4.2.3 Actions

e actions are lower-level procedural constructs that can be used in combination to manipulate the fields of a struct or exchange data with the DUT. Actions can extend over several lines and are separated by semicolons. An action block is a list of actions separated by semicolons and enclosed in curly brackets ({}).

Actions shall be associated with a struct member, specifically a method or an event, or issued interactively as commands at the command line.

The preprocessor commands **#ifdef** and **#ifndef** can be used as actions to control parsing (see 21.2).

Example

Here is an example of an action (an invocation of a method, `transmit()`) associated with an event, `xmit_ready`. Another action, **out()** is associated with the `transmit()` method.

```
struct packet{
    event xmit_ready is rise('top.ready');
    on xmit_ready {transmit();};
    transmit() is {
        out("transmitting packet...");
    };
};
```

The following subclauses highlight particular types of actions.

4.2.3.1 Creating or modifying variables

Action	Description
var	Defines a local variable (see 18.2).
=	Assigns or samples values of fields, local variables, or HDL objects (see 18.3).
op=	Performs a complex assignment (such as add and assign, or shift and assign) of a field, local variable, or HDL object (see 18.4).
force	Forces a Verilog net or wire to a specified value, overriding the value driven from the DUT (see 23.1).
release	Releases the Verilog net or wire that was previously forced (see 23.2).

4.2.3.2 Executing actions conditionally

Action	Description
if then else	Executes an action block if a condition is met or a different action block if it is not (see 20.1.1).
case labeled-case-item	Executes one action block out of multiple action blocks depending on the value of a single expression (see 20.1.2).
case bool-case-item	Evaluates a list of Boolean expressions and executes the action block associated with the first expression that is TRUE (see 20.1.3).

4.2.3.3 Executing actions iteratively

Action	Description
while	Executes an action block repeatedly until a Boolean expression becomes FALSE (see 20.2.1).
repeat until	Executes an action block repeatedly until a Boolean expression becomes TRUE (see 20.2.2).
for each in	For each item in a list that is a specified type, executes an action block (see 20.2.3).
for from to	Executes an action block for a specified number of times (see 20.2.4).
for	Executes an action block for a specified number of times (see 20.2.5).
for each line in file	Executes an action block for each line in a file (see 20.3.1).
for each file matching	Executes an action block for each file in the search path (see 20.3.2).

4.2.3.4 Controlling program flow

Action	Description
break	Breaks the execution of the enclosing loop (see 20.4.1).
continue	Stops execution of the enclosing loop and continues with the next iteration of the same loop (see 20.4.2).

4.2.3.5 Invoking methods and routines

Action	Description
method()	Calls a regular method (see 17.2.3).
tcm()	Calls a TCM (see 17.2.1).
start tcm()	Launches a TCM as a new thread (a parallel process) (see 17.2.2).
calling predefined routines	Calls an <i>e</i> predefined routine (see Clause 26).
compute method() or tcm()	Calls a value-returning method without using the value returned (see 17.2.4).
return	Returns immediately from the current method to the method that called it (see 17.2.5).

4.2.3.6 Emitting an event

Action	Description
emit	Causes a named event to occur (see 10.3.2).

4.2.3.7 Performing time-consuming actions

Action	Description
sync	Suspends execution of the current TCM until the TE succeeds (see 13.1.2).
wait	Suspends execution of the current TCM until a given TE succeeds (see 13.1.3).
all of	Executes multiple action blocks concurrently, as separate branches of a fork. The action following the all of action is reached only when all branches of the all of have been fully executed (see 13.2.1).
first of	Executes multiple action blocks concurrently, as separate branches of a fork. The action following the first of action is reached when any of the branches in the first of has been fully executed. (see 13.2.2).
state machine action	Defines a state machine (see 13.3.2).

4.2.3.8 Generating data item

Action	Description
gen	Generates a value for an item, while considering all relevant constraints (see 9.4.1).

4.2.3.9 Detecting and handling errors

Action	Description
check that	Checks the DUT for correct data values (see 16.2.1).
dut_error()	Defines a DUT error message string (see 16.2.2).
assert	Issues an error message if a specified Boolean expression is not true (see 16.4).
warning()	Issues a warning message (see 16.3.1).
error()	Issues an error message when a user error is detected (see 16.3.2).
fatal()	Issues an error message, halts all activities, and exits immediately (see 16.3.3).
try	Catches errors and exceptions (see 16.3.4).

4.2.3.10 Printing

Action	Description
print	Prints <i>e</i> expressions (see 16.1).

4.2.4 Expressions

Expressions are constructs that combine operands and operators to represent a value. The resulting value is a function of the values of the operands and the semantic meaning of the operators.

A few *e* expressions, such as expressions that restrict the range of valid values of a variable, evaluate to constants at compile time. More typically, expressions are evaluated at runtime, resolved to a value of some type, and assigned to a variable or field of that type. Strict type checking is enforced in *e*.

Each expression shall contain at least one operand, which can be the following:

- Literal value
- Constant
- *e* entity, such as a method, field, list, or struct
- HDL entity, such as a signal

A *compound expression* applies one or more operators to one or more operands.

4.3 Struct hierarchy and name resolution

The following subclauses explain the struct hierarchy of an *e* program and how to reference entities within the program.

4.3.1 Struct hierarchy

Because structs can be instantiated as the fields of other structs, a typical *e* program has many levels of hierarchy. Every *e* program contains several *predefined structs*, as well as *user-defined structs*. Figure 2 shows the partial hierarchy of a typical *e* program. The predefined structs are shown in **bold**.

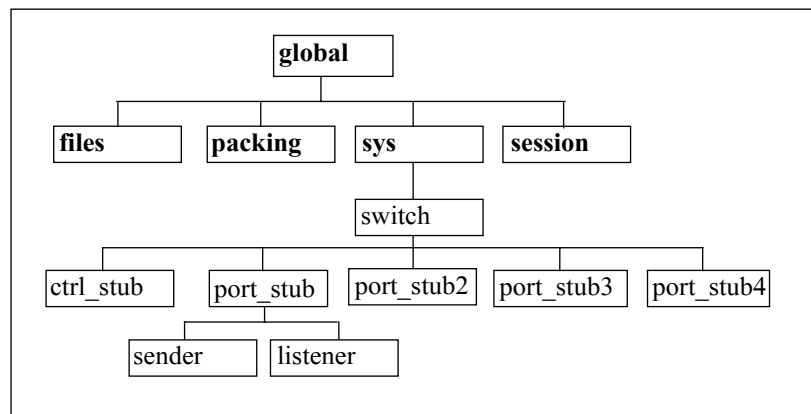


Figure 2—Diagram of struct hierarchy

4.3.1.1 Global struct

The predefined struct **global** is the root of all *e* structs. All predefined structs and most predefined methods are part of the **global** struct. Verification environments shall be nested under **sys**. Extensions to the **global** struct shall be reserved for the implementation of the *e* runtime engine.

The **global** struct shall not be extended.

4.3.1.2 Files struct

The **files** struct provides predefined methods for manipulating files.

4.3.1.3 Packing struct

Packing and unpacking are controlled by a predefined struct under **global** named **packing**. *Packing* and *unpacking* prepare *e* data sent to or received from the DUT. Underneath the **packing** struct are five predefined structs. To create a packing order, copy one of these structs and modify at least one of its parameters.

4.3.1.4 Sys struct

The *system struct* is instantiated under **global** as **sys**. All fields and structs in **sys** not marked by an exclamation point (!) are generated automatically during the **generate_test** phase. Any structs or fields outside of **sys** that need generation shall be generated explicitly.

Time is stored in a 64-bit integer field named **sys.time**. When *e* is linked with an event-driven simulator, **sys.time** shows the current simulator time. When *e* is linked with a cycle-based simulator, **sys.time** shows the current simulator cycle. **sys.time** is influenced by the current timescale.

4.3.1.5 Session struct

The **session** struct holds the status of the current simulator session, related information, and events. Fields in the **session** struct that are of general interest include the following:

- **session.user_time**
- **session.system_time**
- **session.memory**
- **session.check_ok**
- **session.events**

The first three fields listed help determine the time and memory used in a particular session. The next two describe the **check_ok** and **events** fields.

4.3.1.5.1 session.check_ok

This field is of Boolean type and is set to TRUE after every check, if the check succeeds. Otherwise, it is set to FALSE. This field permits behavior checks without the need to duplicate the **if** clause. For example:

```
post_generate() is also {
  check that mlist.size() > 0 else dut_error("Empty list");
  if session.check_ok then {
    check that mlist[0] == 0xa else dut_error("Error at index 0");
  };
};
```

4.3.1.5.2 session.events

This field contains the names of all user-defined events that occurred during the test and how many times each user-defined event occurred. The name of the event is preceded by the struct type and a double underscore, e.g., `struct_type__event_name`.

If an event is defined in a **when** subtype, the name of the event in the **session.events** field is prefixed by the subtype and a double underscore, e.g., `subtype__struct_type__event_name`.

4.3.2 Referencing *e* entities

The following subclauses describe how to reference *e* entities.

4.3.2.1 Structs and fields

Any user-defined struct can be instantiated as a field of any struct. Thus, every *instantiated struct* and its fields have a place in the struct hierarchy and their names include a path reflecting that place. The following considerations also apply:

- The name of the **global** struct can be omitted from the path to a field or a struct.
- The name of the enclosing struct is not included in the path if the current struct is the enclosing struct.
- In certain contexts, the implicit variables **me** or **it** can be used in the path to refer to the enclosing struct. See 4.3.3 for more information.
- If the path begins with a period (.), the path is assumed to start with the implicit variable **it**. See also 4.16.3.
- A special syntax is required to reference struct subtypes and fields under struct subtypes (see 5.1.6).

4.3.2.2 Naming and referencing methods and routines

The names of all methods and routines shall be followed immediately by parentheses, when a method is defined or called. The predefined methods of **any_struct**, such as **pre_generate()** or **init()**, and all user-defined methods, are associated with a particular struct. Thus, like structs and fields, every user-defined method has a place in the struct hierarchy and its name includes a path reflecting that place. User-defined routines, like predefined routines, are associated with the **global** struct. Thus, the term **global** can be omitted from a path when the context is unambiguous (see 4.3.4). See also Clause 24, Clause 25, Clause 26, and Clause 27.

Example 1

This example illustrates the names used to call user-defined and predefined methods.

```
<'  
struct meth {  
    %size: int;  
    %taken: int;  
  
    get_free(size: int, taken: int): int is inline {  
        result = size - taken;};  
};  
extend sys {  
    !area: int;  
    mi: meth;  
  
    post_generate() is also {  
        sys.area = sys.mi.get_free(sys.mi.size, sys.mi.taken);  
        print sys.area;  
    };  
};  
'>
```

Some predefined methods, such as the methods used to manipulate lists, are pseudo-methods. They are not associated with a particular struct. These methods are called by appending their name to the desired expression.

Example 2

Here is an example of how to call the list pseudo-method **.size()**.

```
<'  
  struct meth {  
    %data: list of int;  
  
    keep data.size() <= 10;  
  };  
'>
```

4.3.2.3 Enumerated type values

Names for enumerated type values shall be unique within each type. For example, defining a type as `my_type: [a, a, b]` results in an error because the name `a` is not unique.

However, the same name can be used in more than one enumerated type. For example, the following two enumerated types define the same value names:

```
type destination: [a, b, c, d];  
type source: [a, b, c, d];
```

To refer to an enumerated type value in a struct where no values are shared between the enumerated types, just use the value name. In structs where more than one enumerated field can have the same value, the following syntax shall be used to refer to the value when the type is not clear from the context:

```
type_name'value
```

Example

In the following **keep** constraint, it is clear that the type of `dest` is `destination`, so the value name `b` can unambiguously be used by itself:

```
type destination: [a, b, c, d];  
type source: [a, b, c, d];  
struct packet {  
  dest: destination;  
  keep me.dest == b;
```

However, because the type of the following variable `tmp` is not specified, it is necessary to use the full name for the enumerated type value `destination'b`:

```
type destination: [a, b, c, d];  
type source: [a, b, e, f];  
...  
m() is {  
  var tmp := destination'b;  
};
```

For more about **var** declaration using type inference, see 18.2.

4.3.3 Implicit variables

Many *e* constructs create implicit variables. The scope of these implicit variables is the construct that creates them. Implicit variables can be used in pathnames when referencing *e* entities.

Except for the **result** variable (see 4.3.3.3), values cannot be assigned to implicit variables. An assignment such as `me = packet` shall generate an error.

4.3.3.1 *it*

The implicit variable **it** always refers to the current item. The following constructs create the implicit variable **it**:

- list pseudo-methods (see Clause 24)
- **for each** (see 20.2.3)
- **gen... keeping** (see 9.4.1)
- **keep for each** (see 9.2.7.3.8)
- **keep .is_all_iterations()** (see 9.3.3)
- **new with** (see 4.16.2)
- list with **key** declaration (see 24.7.1)

Wherever an **it**.*field* can be used, the shorthand notation *field* can be used in its place. For example, `it.len` can be abbreviated to `.len`, with a leading dot (`.`). In many places it is legal to designate and use a name other than the implicit **it**.

4.3.3.2 *me*

The implicit variable **me** refers to the current struct and can be used anywhere in the struct. When referring to a field from another member of the same struct, the `me .` can be omitted.

4.3.3.3 *result*

The **result** variable refers to an implicit variable of the method's returned type. It can be assigned within the method body either implicitly or by using the **return** action (see 17.2.5). If no **return** action is encountered, **result** is returned by default. A method that does not have a returned type does not have a **result** implicit variable.

The following method returns the sum of *a* and *b*:

```
sum(a: int, b: int): int is {  
    result = a + b;  
};
```

4.3.3.4 *index*

The **index** variable is a non-negative integer that holds the current index of the item referred to by **it**. The scope of the **index** variable is limited to the action block. The following constructs create the implicit variable **index**:

- list pseudo-methods (see Clause 24)
- **for each** (see 20.2.3)
- **keep for each** (see 9.2.7.3.8)

Example

The following loop assigns 5 to the `len` field of every item in the `packets` list and also assigns the **index** value of each item to its `id` field:

```
for each in packets do {
    packets[index].len = 5;
    .id = index;
};
```

4.3.3.5 prev

The following constraints create the implicit variable **prev**:

- **keep for each** (see 9.2.7.3.8)
- **item ... using also** (see 14.6)

In a **keep for each** context, **prev** refers to the previous item in a list, unless another name is specified. Using **prev** on the first item of the list shall cause an error.

The following shows a sample use of **prev**:

```
keep for each in lof_packet {
    (index > 0) => it.size == prev.size + 1;
};
```

In an **item ... using also** context, extending or changing a **when**, **illegal**, or **ignore** option automatically creates the special variable named **prev**. This variable holds the results of all previous **when**, **illegal**, or **ignore** options, so **prev** can be used as a shorthand to assert those previous options combined with a new option value.

Example

If an original coverage item definition has `when = size == 5`
and an extension has `using also when = (prev and size <= 10)`,

the result is the same as `when = (size == 5 and size <= 10)`.

4.3.4 Name resolution rules

The following subclauses describe how names are resolved, depending on whether they include a path. A *path* is a sequence of names or expressions (returning a string), concatenated with a dot (`.`).

4.3.4.1 Names that include a path

To resolve names that include a path, an entity of that name is searched for at the specified scope. An error message shall be issued if the entity is not found. If the path begins with a period (`.`), the path is assumed to begin with the implicit variable **it**.

4.3.4.2 Names that do not include a path

To resolve names that do not include a path, the following checks are performed, in order. The program stops checking once the named object has been identified.

- a) Check whether the name is a macro. If there are two macro definitions, choose the most recent one.

- b) Check whether the name is one of the predefined constants. It shall not be the same as one of the predefined constants.
- c) Check whether the name is a value of an enumerated type. The value shall uniquely identify an enumerated type.
- d) Check whether the name identifies a variable used in the current action block. If not, and if the action is nested, check whether the name identifies a variable in the enclosing action block. If not, this search continues from the immediately enclosing action block outwards to the boundary of the method.
- e) Check whether the name identifies a member of the current struct:
 - 1) If the expression is inside a struct definition, the current struct is the enclosing struct.
 - 2) If the expression is inside a method, the current struct is the struct to which the method belongs.
- f) Check whether the name identifies a member of the **global** struct.
- g) If the name is still unresolved, an error message shall be issued.

Macros, predefined constants, and enumerated types have a *global scope*, which means they can be seen from anywhere within an *e* program. For that reason, their names shall be unique.

- No two name macros can have the same *name*, and no two replacement macros can have the same *macro-name'nonterminal-type* (see Clause 15).
- No user-defined constant can have the same name as a predefined constant (see 4.1.4.4).
- No two enumerated types can have the same *enum-type-name* (see 5.6).

4.4 Ranges

Ranges are constructs that can be used in specific contexts to specify a range of valid values:

- define scalar subtypes
- field declarations to specify constraints on the generation
- coverage declarations
- **case** action
- bit-slicing
- some expressions

Use the following syntax to restrict the range of valid values:

`[range, ...]`

where *range* is a constant expression or a range of constant expressions in the form:

`low-value..high-value`

Example

```
u: uint[5..7, 15];
```

If the scalar type is an enumerated type, it is ordered by the value associated with the integer value of each type item.

4.5 Operator precedence

Table 12 summarizes all *e* operators in order of precedence. The precedence is the same as in the C language, with the exception of operators that do not exist in C. To change the order of computation, place parentheses around the first expression to compute.

Table 12—Operators in order of precedence

Operator	Operation type
[]	List indexing (subscripting) (see 4.12.1)
[..]	List slicing (see 4.12.3)
[:]	Bit slicing (selection) (see 4.12.2)
f(...)	Method and routine calls (see 4.2.3.5)
Dot operator (.)	Field selection (see 4.16.3)
~, ! (not)	Bitwise not, Boolean not (see 4.7.1, 4.8.1)
{... ; ...}	List concatenation (see 4.12.4)
%{... , ...}	Bit concatenation (see 4.12.5)
Unary + –	Unary plus, minus (see 4.9.1)
*, /, %	Binary multiply, divide, modulus (see 4.9.2)
+, –	Binary add and subtract (see 4.9.2)
>> <<	Shift right, shift left (see 4.7.3)
< <= > >=	Comparison (see 4.10.1)
is [not] a	Subtype identification (see 4.16.1)
== !=	Equality, inequality (see 4.10.2)
=== !==	Verilog four-state comparison (see 4.10.3)
~ !~	String matching (see 4.10.4)
in	Range list operator (see 4.10.5)
&	Bitwise and (see 4.7.2)
	Bitwise or (see 4.7.2)
^	Bitwise xor (see 4.7.2)
&& (and)	Boolean and (see 4.8.2)
(or)	Boolean or (see 4.8.3)
=>	Boolean implication (see 4.8.4)
Conditional operator (? :)	Conditional operator (a ? b : c means “if a then b else c”) (see 4.16.5)

NOTE—Every operation in *e* is performed within the context of types and is carried out with 32-bit precision or unbounded precision (see Clause 5).

4.6 Evaluation order of expressions

In *e*, **and** (&&) and **or** (||) use left-to-right lazy evaluation. Consider the following statement:

```
bool_1 = foo(x) && bar(x)
```

If *foo*(*x*) returns TRUE, then *bar*(*x*) is evaluated as well, to determine whether *bool_1* gets TRUE. If, however, *foo*(*x*) returns FALSE, then *bool_1* gets FALSE immediately and *bar*(*x*) is not executed. The argument to *bar*(*x*) is not even evaluated.

Expressions containing || are likewise evaluated in a lazy fashion: If the sub-expression on the left of the **or** operator is TRUE, then the sub-expression on the right is ignored. Left-to-right evaluation is only required for the operators && or ||.

Now, consider the following statement:

```
bool_2 = foo(x) + bar(x)
```

If *foo*(*x*) or *bar*(*x*) has side effects (i.e., if *foo*(*x*) changes the value of *x* or *bar*(*x*) changes the value of *x*), then the results of *foo*(*x*) + *bar*(*x*) might depend on which of the two sub-expressions, *foo*(*x*) or *bar*(*x*), is evaluated first, so the results are not predictable based on the *e* language definition.

NOTE— Standard-compliant implementations may guarantee an evaluation order to assure predictable results.

4.7 Bitwise operators

The following subclauses describe the *e* bitwise operators. See also 23.1 and 26.3.

4.7.1 ~

Purpose	Unary bitwise negation	
Category	Expression	
Syntax	<i>~exp</i>	
Parameters	<i>~exp</i>	A numeric expression or an HDL pathname.

This sets each 1 bit of an expression to 0 and each 0 bit to 1. Each bit of the resulting expression is the opposite of the same bit in the original expression. When the type and bit-size of an HDL signal cannot be determined from the context, the expression is automatically cast as an unsigned 32-bit integer.

Syntax example:

```
print ~x using hex;
```