

# IEEE P1647/D5 Draft Standard for the Functional Verification Language *e*

Prepared by the *e* Functional Verification Language Working Group (*e*WG)

**Committee**  
of the  
**IEEE Society**

Copyright © 2008-2010 by the IEEE.  
Three Park Avenue  
New York, New York 10016-5997, USA  
All rights reserved.

This document is an unapproved draft of a proposed IEEE Standard. As such, this document is subject to change. **USE AT YOUR OWN RISK!** Because this is an unapproved draft, this document must not be utilized for any conformance/compliance purposes. Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of international standardization consideration. Prior to adoption of this document, in whole or in part, by another standards development organization permission must first be obtained from the IEEE Standards Activities Department (stds.ipr@ieee.org). Other entities seeking permission to reproduce this document, in whole or in part, must also obtain permission from the IEEE Standards Activities Department.

IEEE Standards Activities Department  
445 Hoes Lane  
Piscataway, NJ 08854, USA

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

Namespaces for Types in *e*, Version 1.1

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

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

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

*e* Ports, Chapter 6

Specman Beta Features, Chapter 4 (Constant Fields and Constant when Subtypes)

Specman Beta Features, Chapter 15 (Reflection Interface for *e*)

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

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

---

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

Copyright © 2010 by the Institute of Electrical and Electronics Engineers, Inc.  
All rights reserved. Published xx Month 200x. 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.

PDF: ISBN XXX-XXXXX-XXXX-XSTDXXXXX  
Print: ISBN XXX-XXXXX-XXXX-X STDPDXXXXX

*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 or the soundness of any judgments contained in its standards.

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

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

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. 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 his or her independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

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

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a **proposed change of text**, together with appropriate supporting comments. Comments on standards and requests for interpretations should be submitted to the following address:

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 1647-2010, IEEE Standard for the Functional Verification Language *e*.

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

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

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

The *e* language was productized by Verisity, Ltd., in 1996, as part of a functional verification tool suite. The proliferation of the *e* language and the growing investment in *e*-based intellectual property (IP) compelled the creation of the *e steering committee* in June of 2002, composed of individuals from Texas Instruments, Rambus, 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 (OO) programming model, with implicit memory management and single inheritance. In this, *e* is similar to Java<sup>TM</sup>.<sup>a</sup>
- 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<sup>®</sup><sup>b</sup> (see IEEE Std 1364<sup>TM</sup>)<sup>c</sup> 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.

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

<sup>b</sup>Verilog is a registered trademark of Cadence Design Systems, Inc.

<sup>c</sup>Information on references can be found in Clause 2.

- *e* has many built-in constructs aimed at simplifying common programming tasks; built-in support for lists and hashes; and pattern matching and string- and file-manipulation features, which are borrowed from Perl<sup>TM</sup><sup>d</sup>.
- The *e* syntax is extendable with a powerful macro capability.

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

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

## Notice to users

### Laws and regulations

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

### Copyrights

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

### Updating of IEEE documents

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

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

### Errata

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

<sup>d</sup>Perl is a registered trademark of Perl, Inc.

## Interpretations

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

## Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. A patent holder or patent applicant has filed a statement of assurance that it will grant licenses under these rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses. Other Essential Patent Claims may exist for which a statement of assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

## Participants

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

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

Ajeetha V. Kumari  
 Amy Witherow  
 Andrew Piziali  
 Brett G. Lammers  
 Darren Galpin  
 David G. Von Bank  
 Dean D'Mello

Henry J. Von Bank  
 Iraklis I. Diamantidis  
 J. L. Gray  
 Joseph Hupcey III  
 Kishore Karnane  
 Mark Strickland  
 Matan Vax

Michael G. Bartley  
 Michael T. McNamara  
 Serrie Chapman  
 Srinivasan Venkataramanan  
 Stylianos Diamantidis  
 Tucker Brown  
 Yaron Kashai

## Contents

1.	Overview.....	1
1.1	Scope.....	1
1.2	Purpose.....	1
1.3	Verification environments .....	2
1.4	Basic concepts relating to this standard.....	3
1.5	Conventions used .....	8
1.6	Use of color in this standard .....	10
1.7	Contents of this standard.....	10
2.	Normative references .....	13
3.	Definitions, acronyms, and abbreviations.....	13
3.1	Definitions .....	13
3.2	Acronyms and abbreviations .....	15
4.	<i>e</i> basics .....	17
4.1	Lexical conventions .....	17
4.2	Syntactic elements .....	24
4.3	Struct hierarchy and name resolution .....	30
4.4	Ranges.....	36
4.5	Operator precedence .....	36
4.6	Evaluation order of expressions.....	36
4.7	Bitwise operators .....	38
4.8	Boolean operators .....	39
4.9	Arithmetic operators .....	41
4.10	Comparison operators .....	42
4.11	String matching.....	46
4.12	Extraction and concatenation operators .....	48
4.13	Scalar modifiers .....	52
4.14	Parentheses.....	53
4.15	list.method() .....	53
4.16	Special-purpose operators.....	54
5.	Data types .....	59
5.1	<i>e</i> data types.....	59
5.2	Untyped expressions .....	65
5.3	Assignment rules.....	65
5.4	Real data type.....	68
5.5	Precision rules for numeric operations .....	70
5.6	Automatic type casting .....	72
5.7	Defining and extending scalar types .....	73
5.8	Type-related constructs.....	75
6.	Structs, subtypes, and fields.....	81
6.1	Structs overview .....	81
6.2	Defining structs: struct .....	82

6.3	Extending structs: extend type .....	83
6.4	Restrictions on inheritance.....	83
6.5	Extending subtypes.....	84
6.6	Creating subtypes with when.....	84
6.7	Extending when subtypes .....	86
6.8	Defining fields: field .....	87
6.9	Defining list fields .....	89
6.10	Projecting list of fields .....	91
6.11	Defining attribute fields .....	91
7.	Units.....	93
7.1	Overview.....	93
7.2	Defining units and fields of type unit .....	95
7.3	Unit attributes .....	99
7.4	Predefined methods of any_unit .....	100
7.5	Unit-related predefined methods of any_struct .....	102
7.6	Unit-related predefined routines .....	104
8.	Template types.....	107
8.1	Defining a template type .....	107
8.2	Instantiating a template type .....	109
8.3	Template types and reflection.....	110
8.4	Template errors .....	111
8.5	Limitations .....	112
8.6	Templates vs. macros.....	112
9.	<i>e</i> ports.....	115
9.1	Introduction to <i>e</i> ports.....	115
9.2	Using simple ports .....	116
9.3	Using buffer ports .....	118
9.4	Using event ports .....	119
9.5	Using method ports.....	120
9.6	Defining and referencing ports .....	122
9.7	Port attributes.....	127
9.8	Buffer port methods .....	139
9.9	MVL methods for simple ports.....	141
9.10	Global MVL routines.....	147
9.11	Comparative analysis of ports and tick access.....	150
9.12	<i>e</i> Port Binding .....	152
9.13	TLM Interface Ports in <i>e</i> .....	153
10.	Constraints and generation.....	163
10.1	Types of constraints .....	163
10.2	Generation concepts.....	163
10.3	Type constraints.....	179
10.4	Defining constraints .....	182
10.5	Invoking generation .....	189
11.	Events.....	191



11.1	Causes of events.....	191
11.2	Scope of events .....	191
11.3	Defining and emitting named events .....	191
11.4	Predefined events .....	192
12.	Temporal expressions .....	195
12.1	Overview .....	195
12.2	Temporal operators and constructs .....	198
12.3	Success and failure of a temporal expression .....	213
13.	Temporal struct members .....	215
13.1	on .....	215
13.2	expect   assume .....	216
14.	Time-consuming actions.....	217
14.1	Synchronization actions.....	217
14.2	Concurrency actions .....	218
14.3	State machines .....	220
15.	Coverage constructs.....	225
15.1	Defining coverage groups: cover .....	225
15.2	Defining basic coverage items: item .....	227
15.3	Defining cross coverage items: cross .....	231
15.4	Defining transition coverage items: transition .....	233
15.5	Extending coverage groups: cover ... using also ... is also .....	234
15.6	Extending coverage items: item ... using also .....	235
15.7	Coverage API.....	236
15.8	Coverage methods for the covers struct.....	242
16.	Macros .....	247
16.1	Overview .....	247
16.2	define-as statement .....	248
16.3	define-as-computed statement .....	248
16.4	Match expression structure .....	249
16.5	Interpretation of match expressions .....	252
16.6	Macro expansion code .....	253
17.	Print, checks, and error handling .....	257
17.1	print .....	257
17.2	Handling DUT errors .....	257
17.3	Handling user errors.....	262
17.4	Handling programming errors: assert .....	264
18.	Methods .....	265
18.1	Rules for defining and extending methods .....	265
18.2	Invoking methods .....	273
18.3	Parameter passing .....	276

18.4	Using the C interface .....	278
19.	Creating and modifying <i>e</i> variables .....	281
19.1	About <i>e</i> variables .....	281
19.2	var .....	281
19.3	= .....	282
19.4	op= .....	282
19.5	<= .....	283
20.	Packing and unpacking .....	285
20.1	Basic packing .....	285
20.2	Predefined pack options.....	288
20.3	Customizing pack options.....	289
20.4	Packing and unpacking specific types .....	289
20.5	Implicit packing and unpacking.....	295
21.	Control flow actions.....	297
21.1	Conditional actions .....	297
21.2	Iterative actions.....	299
21.3	File iteration actions.....	303
21.4	Actions for controlling the program flow .....	304
22.	Importing and preprocessor directives.....	307
22.1	Importing <i>e</i> modules .....	307
22.2	#ifdef, #ifndef .....	308
22.3	#define .....	309
22.4	#undef .....	310
23.	Encapsulation constructs.....	311
23.1	package: package-name .....	311
23.2	package: type-declaration .....	311
23.3	package   protected   private: struct-member .....	312
23.4	Scope operator (::) .....	313
24.	Simulation-related constructs .....	315
24.1	force .....	315
24.2	release .....	315
24.3	Tick access: 'hdl-pathname' .....	316
24.4	simulator_command().....	316
24.5	stop_run() .....	317
25.	Messages.....	319
25.1	Overview.....	319
25.2	The message model.....	319
25.3	message and messagef .....	320
25.4	Message loggers.....	322
25.5	Configuring message loggers with constraints .....	323

25.6	Messaging procedural interface (PI) .....	324
25.7	Examples .....	332
26.	Sequences .....	335
26.1	Overview .....	335
26.2	Sequence statement .....	337
26.3	do sequence action .....	339
26.4	Sequence struct types and members .....	340
26.5	BFM-driver-sequence flow diagrams .....	345
27.	List pseudo-methods library .....	349
27.1	Pseudo-methods overview .....	349
27.2	Using list pseudo-methods .....	349
27.3	Pseudo-methods to modify lists .....	349
27.4	General list pseudo-methods .....	358
27.5	Math and logic pseudo-methods .....	372
27.6	List CRC pseudo-methods .....	374
27.7	Keyed list pseudo-methods .....	376
28.	Predefined methods library .....	379
28.1	Predefined methods of sys .....	379
28.2	Predefined methods of any_struct .....	379
28.3	Methods and predefined attributes of unit any_unit .....	382
28.4	Pseudo-methods .....	382
28.5	Coverage methods .....	384
29.	Predefined routines library .....	385
29.1	Deep copy and compare routines .....	385
29.2	Integer arithmetic routines .....	388
29.3	Real arithmetic routines .....	392
29.4	bitwise_op() .....	393
29.5	get_all_units() .....	394
29.6	String routines .....	394
29.7	Output routines .....	403
29.8	Operating system interface routines .....	405
29.9	set_config() .....	408
29.10	Random routines .....	408
30.	Predefined file routines library .....	411
30.1	File names and search paths .....	411
30.2	File handles .....	411
30.3	Low-level file methods .....	411
30.4	General file routines .....	416
30.5	Reading and writing structs .....	422
31.	Reflection API .....	425
31.1	Introduction .....	425
31.2	Type information .....	426

31.3 Aspect information .....	435
31.4 Value query and manipulation.....	439
32. Predefined resource sharing control structs .....	443
32.1 Semaphore methods .....	443
32.2 How to use the semaphore struct .....	444
33. IP protection.....	449
33.1 Encryption.....	449
33.2 Decryption .....	449
33.3 Reflection API .....	450
33.4 Encryption Targets.....	450
Annex A .....	451
Bibliography .....	451
Annex B .....	453
Source code serialization .....	453
Annex C .....	461
Comparison of when and like inheritance .....	461
Annex D .....	471
Name spaces .....	471
Annex E .....	479
Reflection API examples .....	479
Annex F .....	483
Encryption Targets.....	483

# IEEE Standard for the Functional Verification Language *e*

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

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

## 1. Overview

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

### 1.1 Scope

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

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

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

### 1.2 Purpose

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

### 1.3 Verification environments

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

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

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

- It needs to create input stimulus and feed it into the system being verified.
- It needs to collect the output from the system being verified, as well as the state of selected internal nodes.
- It needs to check that the output matches the expectations, based on the functional requirements, the state of the system being verified, and the inputs provided.
- It needs to measure *functional coverage*: the extent to which functions of the system being verified have been exercised by the verification environment.
- It needs to facilitate error identification, isolation, and debug. For that purpose, test environments contain combinational and temporal assertions, as well as various messaging and logging capabilities.
- The verification environment needs to be able to mimic all possible variations and configurations the system being verified might face in practice.
- The verification environment needs to be able to throw a wide variety of error conditions at the system being verified, in order to test error handling and error recovery.
- The verification environment should be easily controllable, to allow steering by the verification engineers.

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

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

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

A standard definition of the *e* language should serve both practicing verification engineers and EDA tool developers. Engineers using *e* to build verification environments and reusable verification components need to ensure the valuable intellectual property (IP) they create can be interpreted by others. Tool developers

need to agree on consistent syntax and semantics to ensure interoperability between tools. These goals are best facilitated by means of an open standard.

## 1.4 Basic concepts relating to this standard

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

### 1.4.1 Fundamental considerations

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

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

### 1.4.2 Organization of *e* programs

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

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

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

### 1.4.3 Modes of execution

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

#### 1.4.3.1 Stand-alone execution

*e* programs can execute independently. In such cases, the program shall use cycle-based semantics. Time shall be modeled by an internal source generating a sequence of discrete time units (*ticks*). The time variable shall be assigned consecutive integer values, starting from 0 (before the initial tick). The advancement of

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

time shall stop when the program requests simulation to stop or for an exception. These conditions are covered under [24.5](#), [29.9](#), and [Clause 17](#).

#### 1.4.3.2 Co-execution with a master simulator

*e* programs can execute in tandem with a *master simulator*. In this case, control is passed back and forth between the *e* runtime engine and the master simulator. The master simulator is typically a commercial Verilog<sup>®</sup>,<sup>2</sup> VHDL, or SystemC<sup>®</sup><sup>3</sup> 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 [11.4.2](#).

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

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

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

#### 1.4.4 Execution phases

The execution of *e* programs goes through the phases listed in [Table 1](#).

**Table 1—Execution phases**

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

<sup>2</sup>Verilog is a registered trademark of Cadence Design Systems, Inc.

<sup>3</sup>SystemC is a registered trademark of Open SystemC Initiative.



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 [28.2](#).

### 1.4.5 Major semantic components

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

#### 1.4.5.1 Types

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

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

The association of instances and types in *e* can be *static* or *dynamic*. Static typing means an instance is associated with a particular type at creation and that association does not change during the life of the instance. Dynamic typing means the association between instance and type can change during the lifetime of the instance. Dynamic inheritance (sometimes called *when inheritance*), enables the creation of instances that adapt during their lifetime to changing conditions—changing their features and functions accordingly. Do not confuse dynamic typing with *casting*, where an instance is converted to a type with compatible features. Nor is dynamic typing the same as up-casting: an object-oriented (OO) concept that allows an object to be accessed as one of its super-types, abstracting away some of the type features.

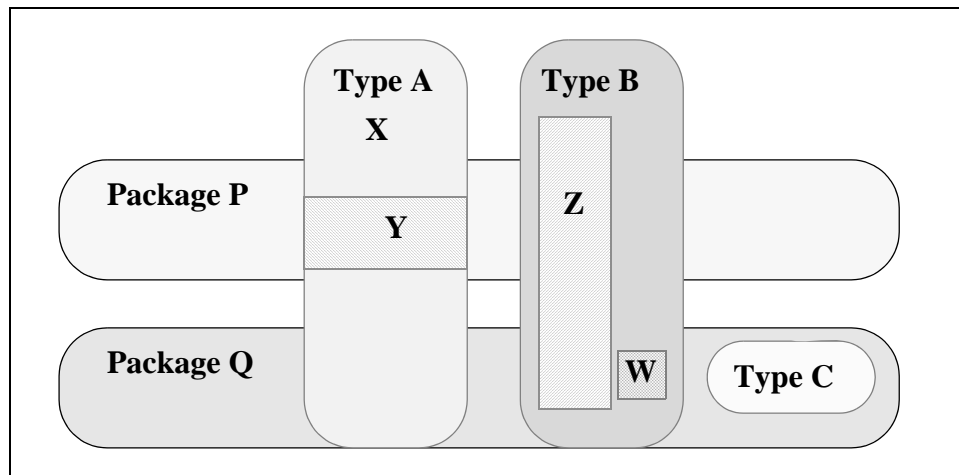
A comprehensive discussion of static and dynamic typing in *e* can be found in [Annex C](#).

#### 1.4.5.2 Packages, aspects, and information hiding

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

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

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



**Figure 1—Type and package membership**

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.

For a detailed discussion of packages and information hiding, see [Clause 23](#).

#### 1.4.5.3 Numeric expressions and values

*e* supports signed and unsigned integer values of arbitrary length. Numbers can be represented as binary, octal, decimal, and hexadecimal values. *e* inherits number representation and much of its arithmetic and logical semantics from the C language (see ISO/IEC 9899).<sup>4</sup> Semantics of specific bit-oriented, logical, and arithmetic operators can be found in [4.7](#), [4.8](#), and [4.9](#), respectively.

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

*e* supports double-precise floating point numbers as the real data type.

#### 1.4.5.4 Constraints and generation

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

<sup>4</sup>For information on references, see [Clause 2](#).

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 10](#).

#### 1.4.5.5 Concurrent execution

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

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

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

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

#### 1.4.5.6 Functional coverage

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

- a) The programmer builds a coverage model, representing key architectural and micro-architectural features of the system being verified.
- b) Coverage data is collected during simulation and accumulated for analysis.
- c) Coverage information is aggregated across many simulations and is analyzed to produce coverage scores. Coverage scores indicate the extent to which the features represented in the coverage models have been exercised by the verification process.

A coverage model includes one or more *cover groups*, which represent a data set to be sampled under certain conditions: the occurrence of a *sampling event* with an optional combinational guard. A cover group consists of *cover items*. Each cover item samples one value for each occurrence of the sampling event and classifies

that value according to programmer-defined categories called *buckets*. During runtime, coverage values are sampled each time a sampling event is emitted. The trail of cumulative samples is the raw data used to determine coverage scores.

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

Functional coverage is distinct from other coverage measures, such as code coverage, state machine coverage, assertion coverage, and the like. However, all these forms of coverage can be combined to form a multi-dimensional hybrid coverage metric. More information about functional coverage can be found in [Clause 15](#) and in *Functional Verification Coverage Measurement and Analysis* [B6].<sup>5</sup>

#### 1.4.5.7 Checking, printing, and reporting

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

*e* programs can contain code to identify, and react to, various error conditions. One source of errors is unexpected behavior of the system being verified. These errors are called *DUT errors* (DUT stands for *design under test*). A second source of errors 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.

The checking, printing, and reporting constructs in *e* are discussed in [Clause 17](#).

### 1.5 Conventions used

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

#### 1.5.1 Visual cues (meta-syntax)

The meta-syntax for the description of lexical and syntax rules uses the conventions shown in [Table 2](#).

#### 1.5.2 Syntax notation within a construct

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

#### *Examples*

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

<sup>5</sup>The numbers in brackets correspond to those of the bibliography in [Annex A](#).

**Table 2—Document conventions**

Visual cue	Represents
courier	The <code>courier</code> font indicates <i>e</i> or HDL code. For example, the following line indicates <i>e</i> code:  <pre>keep opcode in [ADD, ADDI];</pre>
<b>bold</b>	The <b>bold</b> font is used to indicate keywords, text that shall be typed exactly as it appears. See also 1.6. For example, in the following command, the keywords “keep” and “reset_soft” as well as the period and the parentheses shall be typed as they appear: <pre>keep item.reset_soft()</pre>
<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): <pre>keep constraint-bool-exp</pre>
[ ] square brackets	Square brackets indicate optional parameters. For example, in the following construct, the keywords “list of” are optional: <pre>var name: [list of] type</pre>
[ ] bold brackets	Bold square brackets are required. For example, in the following construct, the bold square brackets need to be typed as they appear: <pre>extend enum-type-name: [name, ...]</pre>
<i>construct</i> , ... <i>construct</i> ;	An item followed by a separator character (usually a comma or a semicolon) and an ellipsis is an abbreviation for zero or more elements of the specified type, each separated from the next by the separator character. For example, in the following line zero or more actions may appear between the braces, each separated from the next by a semicolon. <pre>if bool-exp [then] {action; ...}</pre>
( ) parenthesis	Parenthesis ( ( ) ) indicates a grouping, usually of alternative choices. For example, the following line shows the “bits” or “bytes” keywords are possible values for the “type” parameter: <pre>type scalar-type (bits   bytes: num)</pre>
separator bar	The separator bar (   ) character indicates alternative choices. For example, the following line indicates either the “bits” or “bytes” keyword shall be used: <pre>type scalar-type (bits   bytes: num)</pre>

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

This is what the notation means.

- The **list.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)
```

### 1.5.3 Syntax examples

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

### 1.6 Use of color in this standard

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

- Cross references that are hyperlinked to other portions of this standard are shown in [underlined-blue text](#) (hyperlinking works when this standard is viewed interactively as a PDF file).
- Syntactic keywords and tokens in the formal language definitions are shown in **boldface-red text**.

### 1.7 Contents of this standard

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

- [Clause 2](#) provides references to other applicable standards that are assumed or required for this standard.
- [Clause 3](#) defines terms 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](#) describes the principles and usage of *e* templates features. Template types in *e* let you define generic structs and units that are parameterized by type, similar to C++ templates. You can later instantiate them, giving specific types as actual parameters.
- [Clause 9](#) defines *e* unit interfaces and their usage within this standard.
- [Clause 10](#) defines test generation and constraint functions within this standard.
- [Clause 11](#) defines how temporal constructs can be used for specifying and verifying behavior over time in a *e* program.
- [Clause 12](#) defines the syntax and semantics of TEs and describes their usage to track temporal behavior.
- [Clause 13](#) defines two other struct members that can be used for temporal coding of events.
- [Clause 14](#) defines what time-consuming actions are and how to use them.
- [Clause 15](#) describes how to define, extend, and use coverage constructs.
- [Clause 16](#) describes how to modify the grammar of the *e* language.
- [Clause 17](#) shows the various constructs that print an expression, check for errors in the DUT, or add exception handling and diagnostics to an *e* program.
- [Clause 18](#) defines how to declare and use *e* methods within an *e* program.
- [Clause 19](#) describes how to create and assign values to *e* variables.
- [Clause 20](#) shows how to perform basic packing and unpacking of scalars, strings, lists, and structs in this standard.
- [Clause 21](#) describes how to use *e* control flow actions.

- [Clause 22](#) defines how to create dependencies between *e* files and use the preprocessor directives in this standard.
- [Clause 23](#) contains the syntax and descriptions of the *e* statements used to create packages and modify access control.
- [Clause 24](#) describes simulation-related actions, expressions, and routines used.
- [Clause 25](#) describes flexible mechanism used to write text messages to various destinations.
- [Clause 26](#) describes a uniform way to define streams of data items and compose them into verification scenarios.
- [Clause 27](#) describes the pseudo-methods used to work with lists.
- [Clause 28](#) describes all the predefined methods used.
- [Clause 29](#) describes all the predefined routines used.
- [Clause 30](#) contains information about using files and the predefined file routines.
- [Clause 31](#) explains how to access structural information about the type system through the application programming interface (API), also known as the reflection API, and defines the reflection API.
- [Clause 32](#) defines some predefined methods that are useful in controlling time-consuming methods (TCMs) and resource sharing between TCMs.
- [Clause 33](#) describes Intellectual Property (IP) Protection, which aims to shield valuable information contained in an IP without sacrificing the usability of the IP as a whole.
- Annexes. Following [Clause 33](#) are a series of annexes.





## 2. Normative references

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

IEC/IEEE 61691-1-1, Behavioral languages—Part 1: VHDL language reference manual.<sup>6, 7</sup>

IEEE Std 1364<sup>TM</sup>, IEEE Standard for Verilog Hardware Description Language.<sup>8</sup>

ISO/IEC 8859-1, Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin Alphabet No. 1.<sup>9</sup>

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:** One or more program modules that contain 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.

<sup>6</sup>IEC 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/>).

<sup>7</sup>IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (<http://standards.ieee.org/>).

<sup>8</sup>The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

<sup>9</sup>ISO/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** (TE) that determines when the TE 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 temporal expression (TE):** An expression describing the behavior of a system in time, the system comprising a set of variables.

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

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

**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 [5.1.3](#)).

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 <b>0b</b> . 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 <b>0x</b> . 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 <b>0o</b> . An underscore (_) can be added anywhere in the number for readability.	0o66_123
K (multiply by 1024)	A decimal integer followed by a <b>K</b> or <b>k</b> . For example, 32K = 32768.	32K 32k 128k
M (multiply by 1024*1024)	A decimal integer followed by an <b>M</b> or <b>m</b> . 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.”<sup>10</sup>

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*

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

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

<sup>10</sup>Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement the standard.

**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

**4.1.4.3.1 Syntax rules**

- a) A single digit represents four bits in hexadecimal base, three bits in octal base, and one bit in binary base. Similarly, the letters **x**, **z**, **u**, **l**, **h**, **w**, and **n** represent four identical bits (for hexadecimal), three identical bits (for octal), or one bit (for binary). For example, `8'h1x` is equivalent to `8'b0001xxxx`.
- b) If the size of the value is smaller than the width, the value is padded to the left. An MSB of 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.
- c) If the size of the value is larger than the size specified for the list, the value is truncated, leaving the least significant bit (LSB) of the literal.
- d) An underscore can be used for breaking up long numbers to enhance readability. It is legal inside the size and inside the value. It is not legal at the beginning of the literal, between the size and the single quote ( ' ), between the base and the value, and between the single quote ( ' ) and the base.
- e) Decimal literals are not supported.
- f) White space shall not be used as a separator between the width number and base or between the base and the value.
- g) The base and the value are not case-sensitive.
- h) Size and base values need to be specified.
- i) In the context of a Verilog comparison operator (`!==` or `===`) or HDL tick access (e.g., `'data' = 32'bx`), only the 4-value subset is supported (**0**, **1**, **u**, or **x**).
- j) Verilog simulators support only the 4-value logic subset.
- k) An MVL literal of size 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.
- l) The type-casting operations `as_a()` and `is a` do not propagate the context.
- m) If the type of the expression is numeric (based on its context) or if the type cannot be extracted from the context, the default type remains uint.
- n) Syntactically, the same expression can be a numeric type or MVL literal. For example, `1'b1` can represent the number one (1) or a list of MVL with the value `{MVL_1}`.



#### 4.1.4.3.2 Examples

```

32'hffffxxxx
32'HFFFFFFXXX
//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 <b>int</b> ( $2^{31}-1$ )
MIN_INT	Represents the smallest 32-bit <b>int</b> ( $-2^{31}$ ).
MAX_UINT	Represents the largest 32-bit <b>uint</b> ( $2^{32}-1$ ).

#### 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	Newline
\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**.

##### 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 [22.3](#).

**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 an	is c routine	is empty	is first	is inline
is instance	is not a	is not an	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

#### 4.1.5.5 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 <a href="#">4.2.1</a> 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 <a href="#">4.2.2</a> 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 <a href="#">4.2.3</a> 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 <a href="#">4.2.4</a> 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 **package** statements shall appear first, then **import** statements, and then all other statements. Preprocessor directives—including any defines (**#ifdef**, **#ifndef**, **define**, or **define as**)—may precede **import** statements.

[Table 10](#) shows the complete list of *e* statements.

### 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`:

**Table 10—Statements**

Statement	Description
<code>struct</code>	Defines a new data structure (see <a href="#">6.2</a> ).
<code>type</code>	Defines an enumerated data type or scalar subtype (see <a href="#">5.7.1</a> , <a href="#">5.7.2</a> , or <a href="#">5.7.3</a> ).
<code>extend</code>	Modifies a previously defined struct or type (see <a href="#">5.7.4</a> or <a href="#">6.3</a> ).
<code>define as</code>	Extends the <i>e</i> language by defining new statements, struct members, actions, or expressions (see <a href="#">16.2</a> ).
<code>package</code>	Associates a module with a package (see <a href="#">23.1</a> ).
<code>#define, #ifdef, #ifndef, #undef</code>	Place conditions on the <i>e</i> parser (see <a href="#">Clause 22</a> ).
<code>import</code>	Declares dependency between two <i>e</i> modules (see <a href="#">22.1.1</a> ).
<code>unit</code>	Defines a data struct associated with an HDL component or block (see <a href="#">7.2.1</a> ).
<code>C routine,</code> <code>C export</code>	Allows C code to be called from <i>e</i> (see <a href="#">18.4</a> ).

```

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.
Coverage declaration	Defines functional verification goals and collects data on how well the verification is meeting those goals.
Temporal declaration	Defines <i>e</i> events and their associated actions.
<code>#ifdef, #ifndef</code>	Place conditions on the <i>e</i> parser (see <a href="#">22.2</a> ).

### 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 braces (`{ }`).

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 [22.2](#)).

#### *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 <a href="#">19.2</a> ).
=	Assigns or samples values of fields, local variables, or HDL objects (see <a href="#">19.3</a> ).
op=	Performs a complex assignment (such as add and assign, or shift and assign) of a field, local variable, or HDL object (see <a href="#">19.4</a> ).
force	Forces a Verilog net or wire to a specified value, overriding the value driven from the DUT (see <a href="#">24.1</a> ).
release	Releases the Verilog net or wire that was previously forced (see <a href="#">24.2</a> ).

#### 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 <a href="#">21.1.1</a> ).
case labeled-case-item	Executes one action block out of multiple action blocks depending on the value of a single expression (see <a href="#">21.1.2</a> ).
case bool-case-item	Evaluates a list of Boolean expressions and executes the action block associated with the first expression that is TRUE (see <a href="#">21.1.3</a> ).

#### 4.2.3.3 Executing actions iteratively

Action	Description
while	Executes an action block repeatedly until a Boolean expression becomes FALSE (see <a href="#">21.2.1</a> ).
repeat until	Executes an action block repeatedly until a Boolean expression becomes TRUE (see <a href="#">21.2.2</a> ).
for each in	For each item in a list that is a specified type, executes an action block (see <a href="#">21.2.3</a> ).
for from to	Executes an action block for a specified number of times (see <a href="#">21.2.4</a> ).
for	Executes an action block for a specified number of times (see <a href="#">21.2.5</a> ).
for each line in file	Executes an action block for each line in a file (see <a href="#">21.3.1</a> ).
for each file matching	Executes an action block for each file in the search path (see <a href="#">21.3.2</a> ).

#### 4.2.3.4 Controlling program flow

Action	Description
break	Breaks the execution of the enclosing loop (see <a href="#">21.4.1</a> ).
continue	Stops execution of the enclosing loop and continues with the next iteration of the same loop (see <a href="#">21.4.2</a> ).

#### 4.2.3.5 Invoking methods and routines

Action	Description
method()	Calls a regular method (see <a href="#">18.2.3</a> ).
tcm()	Calls a TCM (see <a href="#">18.2.1</a> ).
start tcm()	Launches a TCM as a new thread (a parallel process) (see <a href="#">18.2.2</a> ).
calling predefined routines	Calls an <i>e</i> predefined routine (see <a href="#">Clause 29</a> ).
compute method() or tcm()	Calls a value-returning method without using the value returned (see <a href="#">18.2.4</a> ).
return	Returns immediately from the current method to the method that called it (see <a href="#">18.2.5</a> ).

#### 4.2.3.6 Emitting an event

Action	Description
emit	Causes a named event to occur (see <a href="#">11.3.2</a> ).

#### 4.2.3.7 Performing time-consuming actions

Action	Description
sync	Suspends execution of the current TCM until the TE succeeds (see <a href="#">14.1.2</a> ).
wait	Suspends execution of the current TCM until a given TE succeeds (see <a href="#">14.1.3</a> ).
all of	Executes multiple action blocks concurrently, as separate branches of a fork. The action following the <b>all of</b> action is reached only when all branches of the <b>all of</b> have been fully executed (see <a href="#">14.2.1</a> ).
first of	Executes multiple action blocks concurrently, as separate branches of a fork. The action following the <b>first of</b> action is reached when any of the branches in the <b>first of</b> has been fully executed. (see <a href="#">14.2.2</a> ).
state machine action	Defines a state machine (see <a href="#">14.3.2</a> ).



#### 4.2.3.8 Generating data item

Action	Description
gen	Generates a value for an item, while considering all relevant constraints (see <a href="#">10.5.1</a> ).

#### 4.2.3.9 Detecting and handling errors

Action	Description
expect	Checks the DUT for correct temporal behavior (see <a href="#">13.2</a> ).
check that	Checks the DUT for correct data values (see <a href="#">17.2.1</a> ).
dut_error()	Issues a DUT error message (see <a href="#">17.2.2</a> ).
assert	Checks the verification environment for correct behavior (see <a href="#">17.4</a> ).
warning()	Issues a warning message (see <a href="#">17.3.1</a> ).
error()	Issues an error message when a user error is detected and halts all method execution (see <a href="#">17.3.2</a> ).
fatal()	Issues an error message, halts all activities, and exits immediately (see <a href="#">17.3.3</a> ).
try	Catches errors and exceptions (see <a href="#">17.3.4</a> ).

#### 4.2.3.10 Printing

Action	Description
print	Prints <i>e</i> expressions (see <a href="#">17.1</a> ).

### 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**.

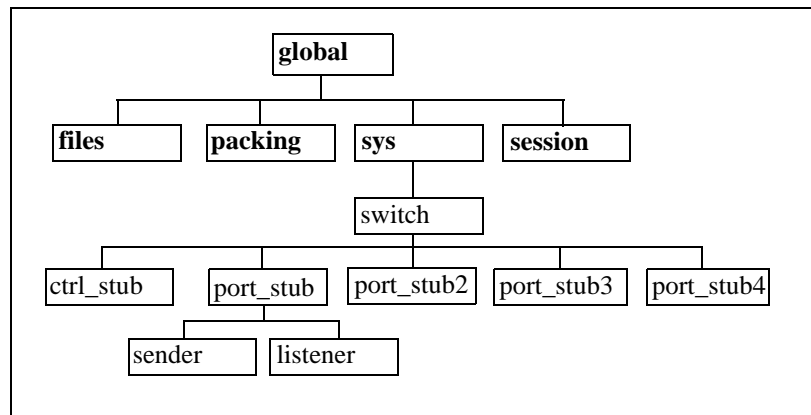


Figure 2—Diagram of struct hierarchy

##### 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 27](#), [Clause 28](#), [Clause 29](#), and [Clause 30](#).

##### *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 [19.2](#).

#### 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 27](#))
- **for each** (see [21.2.3](#))

- **gen ... keeping** (see [10.5.1](#))
- **keep for each** (see [10.2.7.3.8](#))
- **keep .is\_all\_iterations()** (see [10.4.3](#))
- **new with** (see [4.16.2](#))
- list with **key** declaration (see [27.7.1](#))

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 [18.2.5](#)). 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 27](#))
- **for each** (see [21.2.3](#))
- **keep for each** (see [10.2.7.3.8](#))

#### 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 [10.2.7.3.8](#))
- **item ... using also** (see [15.6](#))

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 16](#)).
- 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 [5.7](#)).

## 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.

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.



**Table 12—Operators in order of precedence**

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

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

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

## 4.7 Bitwise operators

The following subclauses describe the  $\epsilon$  bitwise operators. See also [24.1](#) and [29.4](#).

### 4.7.1 ~

<b>Purpose</b>	Unary bitwise negation	
<b>Category</b>	Expression	
<b>Syntax</b>	$\sim exp$	
<b>Parameters</b>	$exp$	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;
```

### 4.7.2 & | ^

<b>Purpose</b>	Binary bitwise operations	
<b>Category</b>	Expression	
<b>Syntax</b>	$exp1 \text{ operator } exp2$	
<b>Parameters</b>	$exp1, exp2$	A numeric expression or an HDL pathname.
	$operator$	$operator$ is one of the following: $\&$ performs an AND operation. $ $ performs an OR operation. $\wedge$ performs an XOR operation.

This performs an AND, OR, or XOR of both operands, bit-by-bit. Operands that are of compatible types, but different lengths, shall be converted to equal types (see [5.5](#)).

Syntax example:

```
print (x & y)
```

### 4.7.3 >> <<

<b>Purpose</b>	Shift bits left or right	
<b>Category</b>	Expression	
<b>Syntax</b>	<i>exp1 operator exp2</i>	
<b>Parameters</b>	<i>exp1</i>	A numeric expression or an HDL pathname.
	<i>operator</i>	<i>operator</i> is one of the following: << performs a shift-left operation. >> performs a shift-right operation.
	<i>exp2</i>	A numeric expression.

This shifts each bit of the first expression to the right or to the left of the number of bits specified by the second expression.

- In a shift-right operation, the shifted bits on the right are lost, while on the left they are filled with 1, if the first expression is a negative integer, or 0 in all other cases.
- In a shift-left operation, the shifted bits on the left are lost, while on the right they are filled with 0.

If the bit-size of the second expression is greater than 32 bits, it is first truncated to 32 bits and then the shift is performed. Truncation removes the MSBs.

Shifting beyond the operand's type (size in bits) is undefined.

Syntax example:

```
outf("%x\n", x >> 4)
```

## 4.8 Boolean operators

The following subclauses describe the *e* Boolean operators. See also [24.3](#).

### 4.8.1 ! (not)

<b>Purpose</b>	Boolean not	
<b>Category</b>	Expression	
<b>Syntax</b>	<b>!</b> <i>exp</i> <b>not</b> <i>exp</i>	
<b>Parameters</b>	<i>exp</i>	A Boolean expression or an HDL pathname.

This returns FALSE when the expression evaluates to TRUE and returns TRUE when the expression evaluates to FALSE.

Syntax example:

```
out(!(3 > 2))
```

#### 4.8.2 && (and)

<b>Purpose</b>	Boolean and	
<b>Category</b>	Expression	
<b>Syntax</b>	<i>exp1</i> && <i>exp2</i> <i>exp1</i> <b>and</b> <i>exp2</i>	
<b>Parameters</b>	<i>exp1, exp2</i>	A Boolean expression or an HDL pathname.

This returns TRUE if both expressions evaluate to TRUE; otherwise, it returns FALSE.

Syntax example:

```
if (2 > 1) and (3 > 2) then {
    out("3 > 2 > 1")
}
```

#### 4.8.3 || (or)

<b>Purpose</b>	Boolean or	
<b>Category</b>	Expression	
<b>Syntax</b>	<i>exp1</i>    <i>exp2</i> <i>exp1</i> <b>or</b> <i>exp2</i>	
<b>Parameters</b>	<i>exp1, exp2</i>	A Boolean expression or an HDL pathname.

This returns TRUE if one or both expressions evaluate to TRUE; otherwise, it returns FALSE.

Syntax example:

```
if FALSE || ('top.a' > 1) then {
    out("'top.a' > 1")
}
```

#### 4.8.4 =>

<b>Purpose</b>	Boolean implication	
<b>Category</b>	Expression	
<b>Syntax</b>	<i>exp1</i> => <i>exp2</i>	
<b>Parameters</b>	<i>exp1, exp2</i>	A Boolean expression.

This returns TRUE when the first expression is FALSE or when the second expression is TRUE. This construct is the same as:

```
(not exp1) or (exp2)
```

See also [10.4.10](#).

Syntax example:

```
out((2 > 1) => (3 > 2))
```

#### 4.8.5 now

<b>Purpose</b>	Boolean event check
<b>Category</b>	Boolean expression
<b>Syntax</b>	<b>now</b> @ <i>event-name</i>
<b>Parameters</b>	<i>event-name</i> The event to check.

This evaluates to TRUE if the event occurs in the same cycle where the **now** expression is encountered, but before the **now** expression is encountered. However, if the event is consumed later during the same cycle, the **now** expression changes to FALSE, i.e., the event can be missed if it succeeds after the expression is encountered. See also [Clause 11](#).

Syntax example:

```
if now @sys.tx_set then {
    out("sys.tx_set occurred")
}
```

## 4.9 Arithmetic operators

The following subclauses describe the *e* arithmetic operators (see also [24.1](#)).

### 4.9.1 Unary + –

<b>Purpose</b>	Unary plus and minus
<b>Category</b>	Expression
<b>Syntax</b>	<i>-exp</i> <i>+exp</i>
<b>Parameters</b>	<i>exp</i> A numeric expression or an HDL pathname.

This performs a unary plus or minus on the expression. The minus operation changes a positive integer to a negative one and a negative integer to a positive one. The plus operation leaves the expression unchanged.

Syntax example:

```
out(5, " == ", +5)
```

#### 4.9.2 + − \* / %

<b>Purpose</b>	Binary arithmetic	
<b>Category</b>	Expression	
<b>Syntax</b>	<i>exp1 operator exp2</i>	
<b>Parameters</b>	<i>exp1, exp2</i>	A numeric expression or an HDL pathname.
	<i>operator</i>	<i>operator</i> is one of the following: <div style="margin-left: 20px;"> + performs addition.  − performs subtraction.  * performs multiplication.  / performs division and returns the quotient, rounded down.  % performs division and returns the remainder. </div>

This performs binary arithmetic operations (see also [29.2](#)).

Syntax example:

```
out(10 + 5)
```

### 4.10 Comparison operators

The following subclauses describe the *e* comparison operators (see also [24.3](#)).

#### 4.10.1 < <= > >=

<b>Purpose</b>	Comparison of values	
<b>Category</b>	Expression	
<b>Syntax</b>	<i>exp1 operator exp2</i>	
<b>Parameters</b>	<i>exp1, exp2</i>	A numeric expression or an HDL pathname.
	<i>operator</i>	<i>operator</i> is one of the following: <div style="margin-left: 20px;"> &lt; returns TRUE if the first expression is smaller than the second expression.  &lt;= returns TRUE if the first expression is not larger than the second expression.  &gt; returns TRUE if the first expression is larger than the second expression.  &gt;= returns TRUE if the first expression is not smaller than the second expression. </div>

This compares two expressions.

Syntax example:

```
print 'top.a' >= 2
```

**4.10.2 == !=**

<b>Purpose</b>	Equality of values	
<b>Category</b>	Expression	
<b>Syntax</b>	<i>exp1 operator exp2</i>	
<b>Parameters</b>	<i>exp1, exp2</i>	A numeric, Boolean, string, list, or struct expression.
	<i>operator</i>	<i>operator</i> is one of the following: <b>==</b> returns TRUE if the first expression evaluates to the same value as the second expression. <b>!=</b> returns TRUE if the first expression does not evaluate to the same value as the second expression.

The equality operators compare the items and return a Boolean result. All types of items are compared by value, except for structs that are compared by address. The following considerations also apply:

- Enumerated type values can be compared as long as they are of the same type.
- Do not use these operators to compare a string to a regular expression. Use the `~` or `!~` operator instead.
- See [4.10.3](#) for a description of using this operator with HDL pathnames.

Comparison methods for the various data types are listed in [Table 13](#).

**Table 13—Equality comparisons for various data types**

Type	Comparison method
integers, unsigned integers, Booleans, HDL pathnames	Values are compared.
strings	The strings are compared character-by-character (case-sensitive).
lists	The lists are compared item-by-item. The list item types need to be compatible; otherwise, an error shall occur.
structs	The structs' addresses are compared

Syntax example:

```
print lob1 == lob2;
print p1 != p2
```

### 4.10.3 === !==

<b>Purpose</b>	Verilog-style four-state comparison operators
<b>Category</b>	Expression
<b>Syntax</b>	<i>'HDL-pathname'</i> [ <b>!==</b>   <b>===</b> ] <i>exp</i> <i>exp</i> [ <b>!==</b>   <b>===</b> ] <i>'HDL-pathname'</i>
<b>Parameters</b>	<i>HDL-pathname</i> The full pathname of an HDL object, this can also include expressions and composite data. See <a href="#">24.3</a> for more information.
	<b>!==</b> Determines non-identity, as in Verilog. Returns TRUE if the left and right operands differ in at least one bit (considering also the x and z values).
	<b>===</b> Determines identity, as in Verilog. Returns TRUE if the left and right operands have identical values (considering also the x and z values).
	<b>! =</b> Returns TRUE if the left and right operands are equal after translating all x values to 0 and all z values to 1.
	<b>= =</b> Returns TRUE if the left and right operands are non-equal after translating all x values to 0 and all z values to 1.
	<i>exp</i> A literal with four-state values, a numeric expression, or another HDL pathname.

This compares four-state values (0, 1, x, and z) with the identity and non-identity operators (Verilog-style operators). The regular equal and non-equal operators can also be used.

There are three ways to use the identity (===) and non-identity (!==) operators, as follows:

- 'HDL-pathname'* == *literal-number-with-x-and-z-values*  
This expression compares an HDL object to a literal number, e.g., `'top.reg' === 4'b11z0`. It checks that the bits of the HDL object match the literal number, bit-by-bit (considering all four values 0, 1, x, and z).
- 'HDL-pathname'* == *number-exp*  
This expression evaluates to TRUE if the HDL object is identical in each bit value to the integer expression *number-exp*. Integer expressions in *e* cannot hold x and z values; thus, the whole expression can be true only if the HDL object has no x or z bits and is otherwise equal to the integer expression.
- 'HDL-pathname'* == *'second-HDL-pathname'*  
This expression evaluates to TRUE if the two HDL objects are identical in all their bits (considering all four values 0, 1, x, and z).

As in Verilog, if the radix is not binary, the z and x values in a literal number are interpreted as more than one bit wide and are left-extended when they are the left-most literal. The width they assume depends on the radix, e.g., in a hexadecimal radix, each literal z counts as four z bits.

Syntax example:

```
// Test a single bit to determine its current state
case {
  'TOP.write_en' === 1'b0 : {out("write_en is 0")};
  'TOP.write_en' === 1'b1 : {out("write_en is 1")};
  'TOP.write_en' === 1'bx : {out("write_en is x")};
}
```



```

    'TOP.write_en' === 1'bz : {out("write_en is z")}
}

```

#### 4.10.4 ~ !~

<b>Purpose</b>	String matching
<b>Category</b>	Expression
<b>Syntax</b>	<i>“string” operator “pattern-string”</i>
<b>Parameters</b>	<i>string</i> A legal <i>e</i> string.
	<i>operator</i> <i>operator</i> is one of the following: ~        returns TRUE if the pattern string can be matched to the whole string. !~      returns TRUE if the pattern string cannot be matched to the whole string.
	<i>pattern-string</i> An AWK-style regular expression or a native <i>e</i> regular expression. If the pattern string starts and ends with slashes (/), then everything inside the slashes is treated as an AWK-style regular expression (see <a href="#">4.11</a> ).

This matches a string against a pattern. There are two styles of string matching: native *e* style, which is the default, and AWK-style. See also [4.11](#).

After a match using either of the two styles, the local pseudo-variable \$0 holds the whole matched string and the pseudo-variables \$1, \$2, ..., \$27 hold the substrings matched. The pseudo-variables are set only by the ~ operator and are local to the function that does the string match. If the ~ operator produces fewer than 28 substrings, the unused variables are left empty.

Syntax example:

```

print s ~ "blue*";
print s !~ "/^Bl.*d$/ "

```

#### 4.10.5 in

<b>Purpose</b>	Check or constrain a value in a list or range	
<b>Category</b>	Expression	
<b>Syntax</b>	<i>exp1</i> <b>in</b> <i>exp2</i>	
<b>Parameters</b>	<i>exp1</i>	<p>When the second expression is a range list, e.g., in a <b>keep</b> constraint, then the type of the first expression has to be of a type comparable to the type of the range list. For a range list, square brackets (<b>[ ]</b>) are used.</p> <p>When the second expression is a list, e.g., in a <b>check</b>, then the type of the first expression can be one of the following:</p> <ul style="list-style-type: none"> <li>— a type that is comparable to the element type of the second expression</li> <li>— a list of type that is comparable to the element type of the second expression.</li> </ul> <p>For a list, braces (<b>{ }</b>) are used.</p>
	<i>exp2</i>	A list or a range list. A <i>range list</i> is a list of constants or expressions that evaluate to constants. Expressions that use variables or struct fields cannot appear in range lists.

For a check, this evaluates TRUE if the first expression is included or contained in the second expression. For a constraint, this designates the range for the first expression.

When two lists are compared and the first list has more than one repetition of the same value (e.g., in { 1 ; 2 ; 1 }, 1 is repeated twice), then at least the same number of repetitions has to exist in the second list for the operator to succeed. See also [4.13.2](#) and [5.1.8](#).

Syntax example:

```
keep x in [1..5];
check that x in {1; 2; 3; 4; 5}
```

## 4.11 String matching

There are two styles of string matching: native *e* style, which is the default, and an AWK-like style. If the pattern starts and ends with slashes (/), then everything inside the slashes is treated as an AWK-style regular expression. See also [29.6](#).

### 4.11.1 Native *e* string matching

Native *e* string matching is attempted on all patterns that are not enclosed in slashes (/). Native style string matching is case-insensitive.

Native style string matching always matches the full string to the pattern. For example: *r* does not match Bluebird, but *\*r\** does. A successful 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.

Native string matching uses the meta-characters shown in [Table 14](#).

*Example*

**Table 14—Meta-characters in native string matching**

Character string	Meaning
" " (blank)	Any sequence of white space (blanks and tabs).
*	Any sequence of non-white space characters, possibly empty (""). a* matches a, ab, and abc, but not "ab c".
...	Any sequence of characters.

The following print statements:

```

m() is {
    var x := "pp kkk";

    print x ~ "* *";
    print $1; print $2;
    print x ~ "...";
    print $1
}

```

produce these results:

```

x ~ "* *" = TRUE
$1 = "pp"
$2 = "kkk"
x ~ "..." = TRUE
$1 = "pp kkk"

```

#### 4.11.2 AWK-style string matching

In AWK-style string matching, a richer regular expression notation can be used to write complex patterns. This notation uses the `/.../` format for the pattern to specify AWK-style regular expression syntax. AWK style supports the special characters shown in [Table 15](#).

The shorthand notations shown in [Table 16](#) (each representing a single character) can also be used in AWK-style regular expressions.

**Table 15—Special characters**

Character	Meaning
.	This matches any single character.
*	The preceding item is matched zero or more times.
+	The preceding item is matched one or more times.
?	The preceding item is optional and matches once, at most.
^	This matches the empty string at the beginning of a line.
\$	This matches the empty string at the end of a line.
<	This matches the empty string at the beginning of a word.

**Table 15—Special characters (*continued*)**

Character	Meaning
>	This matches the empty string at the end of a word.
[ ]	This is a list of options (enclosed in brackets) that matches any character in the list. The list can contain ranges, specified by using a dash (-), e.g., [0-9]. The list can also be negated by using a caret (^), e.g., [^0-9].
\	This is used to escape meta-characters.

**Table 16—Shorthand notations**

Shorthand notation	Meaning
`	A shortest match operator: ` (back tick)
\d	Digit: [0-9]
\D	Non-digit
\s	Any white-space single char
\S	Any non-white-space single
\w	Word char: [a-z A-Z 0-9 _]
\W	Non-word char

After doing a match, the local pseudo-variables \$1, \$2, ..., \$27 correspond to the parenthesized pieces of the match. \$0 stores the whole matched piece of the string.

#### *Example*

The following print statements:

```
m() is {
    var x := "pp--kkk";

    print (x ~ "--/");
    print (x ~ "^pp--kkk$/")
}
```

produce these results:

```
x ~ "--/" = TRUE
x ~ "^pp--kkk$/ " = TRUE
```

## **4.12 Extraction and concatenation operators**

The following subclauses describe the *e* extraction and concatenation operators.

**4.12.1 [ ]**

<b>Purpose</b>	List index operator	
<b>Category</b>	Expression	
<b>Syntax</b>	<i>list-exp</i> <b>[</b> <i>exp</i> <b>]</b>	
<b>Parameters</b>	<i>list-exp</i>	An expression that returns a list.
	<i>exp</i>	A numeric expression.

This extracts or sets a single item from a list. Indexing is only allowed for lists. To get a single bit from a scalar, use bit extraction (see [4.12.2](#)). See also [Clause 27](#).

Syntax example:

```
ints[size] = 8
```

**4.12.2 [ : ]**

<b>Purpose</b>	Select bits or bit slices of an expression	
<b>Category</b>	Expression	
<b>Syntax</b>	<i>exp</i> <b>[</b> <i>[high-exp]</i> <b>:</b> <i>[low-exp]</i> <b>[</b> <i>:slice</i> <b>]</b> <b>]</b>	
<b>Parameters</b>	<i>exp</i>	A numeric expression, an HDL pathname, or an expression returning a list of bit or a list of byte.
	<i>high-exp</i>	A non-negative numeric expression. The high expression shall be greater than or equal to the low expression. To extract a single slice, use the same expression for both the high expression and the low expression. The default value depends on the size of the <i>exp</i> . For example, if <i>exp</i> is a 32-bit integer and the <i>slice</i> is bit, the default value is 32.
	<i>low-exp</i>	A non-negative numeric expression, less than or equal to the high expression. The default value is 0.
	<i>slice</i>	Can be <b>bit</b> , <b>byte</b> , <b>int</b> , or <b>uint</b> . The default is <b>bit</b> .

This extracts or sets consecutive bits or slices of a scalar, a list of bits, or a list of bytes.

When used on the left-hand side (LHS) of an assignment operator, the bit extract operator sets the specified bits of a scalar, a list of bits, or a list of bytes to the value on the right-hand side (RHS) of the operator. The RHS value is chopped or zero/sign extended, if needed. When used in any context except the LHS of an assignment operator, the bit extract operator extracts the specified bits of a scalar, a list of bits, or a list of bytes.

Syntax example:

```
print u[15:0] using hex
```

#### 4.12.2.1 Slice and size of the result

The slice parameter affects the size of the slice that is set or extracted. With the default slice (**bit**), the bit extract operator always operates on a 1-bit slice of the expression. When extracting from a scalar expression, by default, the bit extract operator returns an expression that is the same type and size as the scalar expression. When extracting from a list of bit or a list of byte, by default, the result is a positive unbounded integer.

The bit operator can operate on a larger number of bits when a different slice (**byte**, **int**, or **uint**) is set. For example, the following first print statement displays the lower two bytes of `big_i`, 4096; the second displays the higher 32-bit slice of `big_i`, -61440.

```
var big_i : int (bits:64) = 0xffff1000ffff1000;

print big_i[1:0:byte];
print big_i[1:1:int]
```

#### 4.12.2.2 Accessing nonexistent bits

If the expression is a numeric expression or an HDL pathname, any reference to a non-existent bit shall cause an error. However, for unbounded integers, all bits logically exist: 0 for positive numbers and 1 for negative numbers.

The  $[high : low]$  order of the bit extract operator is the opposite of the  $[low..high]$  order of the list extract operator.

The bit extract operator has a special behavior in packing. Packing the result of a bit extraction uses the exact size in bits ( $high - low + 1$ ). The size of the following pack expression is  $(5 - 3 + 1) + (i - 3 + 1)$ :

```
pack(packing.low, A[5:3], B[i:3])
```

See also 23.3.

#### 4.12.3 [ .. ]

<b>Purpose</b>	List slicing operator	
<b>Category</b>	Expression	
<b>Syntax</b>	<code>exp[[<i>low-exp</i>..<i>high-exp</i>]]</code>	
<b>Parameters</b>	<i>exp</i>	An expression returning a list or a scalar.
	<i>low-exp</i>	An expression evaluating to a non-negative integer. The default is 0.
	<i>high-exp</i>	An expression evaluating to a non-negative integer. The default is the expression size in bits-1.

This accesses the specified list items and returns a list of the same type as the expression. If the expression is a list of bits, it returns a list of bits. If the expression is a scalar, it is implicitly converted to a list of bits.

The rules for the list slicing operator are as follows:

- A list slice of the form `a[m..n]` requires that  $n \geq m \geq 0$  and  $n < a.size()$ . The size of the slice in this case is  $n - m + 1$ .

- b) A list slice of the form `a[m..]` requires that  $m \geq 0$  and  $m \leq a.size()$ . The size of the slice in this case is `a.size() - m`.
- c) A list slice of the form `a[..n]` requires that  $0 \leq n \leq a.size() - 1$ . The size of the slice in this case is `n + 1`.
- d) When assigning to a slice, the size of the RHS shall be the same as the size of the slice; specifically, when the slice is of form `a[m..]` and  $m == a.size()$ , then the RHS shall be an empty list.
- e) The only times a list slice operation returns an empty list is
  - 1) in using `a[m..]`, where  $m == a.size()$ .
  - 2) when the list slice operation is performed on an empty list.
- f) This operator is not supported for unbounded integers.

These rules are also true for the case of list slicing a numeric value. See also [24.3](#).

Syntax example:

```
print packets[0..14]
```

#### 4.12.4 {... ; ...}

<b>Purpose</b>	List concatenation	
<b>Category</b>	Expression	
<b>Syntax</b>	<code>{<i>exp</i>; ...}</code>	
<b>Parameters</b>	<i>exp</i>	Any legal <i>e</i> expression, including a list. All expressions need to be compatible with the result type.

This returns a list built out of one or more elements or other lists. The result type is determined by the following rules:

- The type is derived from the context.
- The type is derived from the first element type of the list.

See also [5.1.8](#).

Syntax example:

```
var x : list of uint = {1; 2; 3}; // list of uint
var y := {50'1; 2; 3}           // list of int 50 bits wide
```

#### 4.12.5 %{... , ...}

<b>Purpose</b>	Bit concatenation operator	
<b>Category</b>	Expression	
<b>Syntax</b>	<b>%{<i>exp1</i>, <i>exp2</i>, ...}</b>	
<b>Parameters</b>	<i>exp1</i> , <i>exp2</i>	Expressions that receive lists of bits (when on the LHS of an assignment operator) or supply lists of bits (when on the RHS of an assignment operator).

This creates a list of bits from two or more expressions, or creates two or more smaller lists of bits from a given expression. Bit concatenations are untyped expressions. In many cases, the required type can be deduced from the context of the expression. See also [5.2](#) and [Clause 20](#).

The bit concatenation operator **%{}** can also be used for packing or unpacking operations that require the **packing.high** order, e.g.,

*value-exp* = **%{*exp1*, *exp2*, ...}** is equivalent to *value-exp* = **pack(packing.high, *exp1*, *exp2*, ...)**  
**%{*exp1*, *exp2*, ...}** = *value-exp* is equivalent to **unpack(packing.high, *value-exp*, *exp1*, *exp2*, ...)**

Syntax example:

```
num1 = %{num2, num3};
%{num2, num3} = num1
```

### 4.13 Scalar modifiers

A scalar subtype can be created by using a scalar modifier to specify the range or bit width of a scalar type. Composing scalar modifiers is allowed. In case of multiple modifiers, the latest bit-size and range shall take effect.

#### 4.13.1 [ range,...]

<b>Purpose</b>	Range modifier	
<b>Category</b>	Expression	
<b>Syntax</b>	<b>[<i>range</i>, ...]</b>	
<b>Parameters</b>	<i>range</i>	A constant expression or a range of constant expressions in the form: low-value..high-value If the scalar type is an enumerated type, it is ordered by the value associated with the integer value of each type item.

This creates a scalar subtype by restricting the range of valid values. Adding a range modifier is equivalent to adding a constraint restricting the field's value to the range.

Syntax example:

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



#### 4.13.2 (bits | bytes : width-exp)

<b>Purpose</b>	Define a sized scalar
<b>Category</b>	Expression
<b>Syntax</b>	<b>(bits   bytes: width-exp)</b>
<b>Parameters</b>	<i>width-exp</i> A positive constant expression.

A scalar subtype can be created by using a scalar modifier to specify bit width of a scalar type. This expression defines a bit width for a scalar type. The actual bit width is  $exp * 1$  for bits and  $exp * 8$  for bytes.

In the following syntax example, both the `word` and `address` types have a bit width of 16:

Syntax example:

```
type word      : uint(bits:16);
type address   : uint(bytes:2)
```

#### 4.14 Parentheses

Parentheses (()) can be used freely to group terms in expressions or to improve the readability of the code, as has been done in some examples in this standard. Parentheses are only required in a few places in *e* code, such as at the end of the method or routine name in all method definitions, method calls, or routine calls. Required parentheses are shown in **boldface** in the syntax listings in this standard.

Parentheses are also required to invoke any method or routine.

#### 4.15 list.method()

<b>Purpose</b>	Execute list pseudo-method
<b>Category</b>	Expression
<b>Syntax</b>	<i>list-exp</i> . <b>list-method</b> ( <b>[param,]</b> )...
<b>Parameters</b>	<i>list-exp</i> An expression that returns a list.
	<i>list-method</i> One of the list pseudo-methods described in <a href="#">Clause 27</a> .

This executes a list pseudo-method on the specified list expression, item-by-item, as follows:

- When an item is evaluated, **it** stands for the item and **index** stands for its index in the list.
- When a parameter is passed, that expression is evaluated for each item in the list.
- List method calls can be nested within any expression, as long as the returned type matches the context.

Syntax example:

```
print me.my_list.is_empty()
```

## 4.16 Special-purpose operators

The following special-purpose operators are supported.

### 4.16.1 **is [not] a, is [not] an**

<b>Purpose</b>	Identify the subtype of a struct instance
<b>Category</b>	Boolean expression
<b>Syntax</b>	<i>struct-exp</i> <b>is</b> ( <b>a</b>   <b>an</b> ) <i>subtype</i> [( <i>name</i> )] <i>struct-exp</i> <b>is not</b> ( <b>a</b>   <b>an</b> ) <i>subtype</i>
<b>Parameters</b>	<i>struct-exp</i> An expression that returns a struct.
	<i>subtype</i> A subtype of the specified struct type.
	<i>name</i> The name of the local variable to create.

This identifies whether a struct instance is a particular subtype or not at runtime. If a name is specified, a local temporary variable of that name is created in the scope of the action containing the **is a** expression. This local variable contains the result of *struct-exp.as\_a(type)* when the **is a** expression returns TRUE. The following considerations also apply:

- A compile-time error shall occur if there is no chance that the struct instance is of the specified type.
- Unlike other constructs with optional *name* variables, the implicit **it** variable is not created when the optional *name* is not used in the **is a** expression.

See also [5.8.1](#).

NOTE—The **is a** and **is an** expressions can be used interchangeably (just as **is not a** and **is not an** can) to make *e* code more like English.

Syntax example:

```

if me is a long packet (lp) then {
    print lp
};

if me is not an extra packet then {
    print kind
}

```

#### 4.16.2 new

<b>Purpose</b>	Allocate a new initialized struct
<b>Category</b>	Expression
<b>Syntax</b>	<b>new</b> [ <i>struct-type</i> [( <i>name</i> )] <b>with</b> { <i>action</i> ; ...}]
<b>Parameters</b>	<i>struct-type</i> A struct type or struct subtype.
	<i>name</i> An optional name, valid only within the action block, for the new struct. If no name is specified, the implicit variable <b>it</b> can be used to reference the new struct.
	<i>action</i> A list of one or more actions.

This creates a new struct, as follows:

- It allocates space for the struct.
- It assigns default values to struct fields.
- It invokes the **init()** method for the struct, which initializes all fields of scalar type, including enumerated scalar type, to zero (0). The initial value of a struct or list is **NULL**, unless the list is a sized list of scalars, in which case it is initialized to the proper size with each item set to the default value.
- It invokes the **run()** method for the struct, unless the **new** expression is in a construct that is executed before the run phase, e.g., if **new** is used in an extension to **sys.init()**, then the **run()** method is not invoked.
- It executes the action block, if one is specified.

The new struct is a shallow struct. The fields of the struct that are of type **struct** are not allocated. If no subtype is specified, the type is derived from the context, e.g., if the new struct is assigned to a variable of type **packet**, the new struct is of type **packet**.

If the optional **with** clause is used, the newly created struct can be referenced with the implicit variable **it** or by using the (optional) *name*.

See also [28.2.2.1](#) and [28.2.2.4](#).

Syntax example:

```
var q : packet = new packet_s
```

### 4.16.3 Dot operator (.)

<b>Purpose</b>	Refer to fields in structs
<b>Category</b>	Expression
<b>Syntax</b>	$[[struct\text{-}exp].] field\text{-}name$ $[[struct\text{-}exp].] event\text{-}name$ $[[struct\text{-}exp].] method\text{-}name$
<b>Parameters</b>	<i>struct-exp</i> An expression that returns a struct or a list of structs.
	<i>field-name</i> The name of the field to reference.
	<i>event-name</i> The name of the event to reference.
	<i>method-name</i> The name of the method to reference.

This refers to a field in the specified struct. If the *struct-exp* is a struct expression, it returns the field in the specified struct. If the *struct-exp* is a list of structs expression, it returns a list containing the contents of the specified *field-name* from all structs in the list. If the *field-name* is a list item, the expression returns a concatenation of the lists in the field.

If the struct expression is missing, but the period exists, the implicit variable **it** is assumed. If both the struct expression and the period (.) are missing, the field name is resolved according to the name resolution rules (see [4.3](#)).

When the struct expression is a list of structs, the expression cannot appear on the LHS of an assignment operator.

Syntax example:

```
keep soft port.sender.cell.u == 0xFF
```

### 4.16.4 Apostrophes (')

The apostrophe (') is an important syntax element used in multiple ways in *e* source code. The actual context of where it is used in the syntax defines its purpose. A single apostrophe is used in the following places:

- When accessing HDL objects (e.g., 'top.a')
- When defining the name of a syntactic construct in a macro definition (e.g., show\_time'command)
- When referring to struct subtypes (e.g., b'dest Ethernet packet)
- When referring to an enumerated value not in context of an enumerated variable (e.g., color'green)
- In the begin-code marker '<' and end-code marker '>'
- In sized numbers (e.g., 2'b11)
- In MVL literals (e.g., 2'bxx)

See also [5.1.6](#) and [Clause 16](#).

#### 4.16.5 Conditional operator (? :)

<b>Purpose</b>	Conditional operator	
<b>Category</b>	Expression	
<b>Syntax</b>	<i>bool-exp ? exp1 : exp2</i>	
<b>Parameters</b>	<i>bool-exp</i>	A legal <i>e</i> expression that evaluates to TRUE or FALSE.
	<i>exp1, exp2</i>	A legal <i>e</i> expression.

This evaluates one of two possible expressions, depending on whether the Boolean expression evaluates to TRUE or FALSE. If the Boolean expression is TRUE, the first expression is evaluated. If it is FALSE, the second expression is evaluated.

See also [21.1](#).

Syntax example:

```
z = (flag ? 7 : 15)
```



## 5. Data types

The *e* language has a number of predefined data types, including the integer and Boolean scalar types common to most programming languages. In addition, new scalar data types (*enumerated types*) that are appropriate for programming, modeling hardware, and interfacing with hardware simulators can be created. The *e* language also provides a powerful mechanism for defining OO hierarchical data structures (*structs*) and ordered collections of elements of the same type (*lists*). The following subclauses provide a basic explanation of *e* data types.

### 5.1 *e* data types

Most *e* expressions have an explicit data type, as follows:

- Scalar types
- Scalar subtypes
- Enumerated scalar types
- Casting of enumerated types in comparisons
- Struct types
- Struct subtypes
- Referencing fields in when constructs
- List types
- The string type
- The real type
- The `external_pointer` type
- The ‘untyped’ pseudo type

Certain expressions, such as HDL objects, have no explicit data type. See [5.2](#) for information on how these expressions are handled.

#### 5.1.1 Scalar types

Scalar types in *e* are one of the following: numeric, Boolean, or enumerated. [Table 17](#) shows the predefined numeric and Boolean types.

Both signed and unsigned integers can be of any size and, thus, of any range. See [5.1.2](#) for information on how to specify the size and range of a scalar field or variable explicitly. See also [Clause 4](#).

#### 5.1.2 Scalar subtypes

A *scalar subtype* can be named and created by using a scalar modifier to specify the range or bit width of a scalar type. Unbounded integers are a predefined scalar subtype. The following subclauses describe scalar modifiers, named scalar subtypes, and unbounded integers in more detail.

##### 5.1.2.1 Scalar modifiers

There are two types of scalar modifiers that can be used to modify predefined scalar types:

- Range modifiers
- Width modifiers

**Table 17—Predefined scalar types**

Type name	Function	Default size for packing	Default value
<b>int</b>	Represents numeric data, both negative and non-negative integers.	32 bits	0
<b>uint</b>	Represents unsigned numeric data, non-negative integers only.	32 bits	0
<b>bit</b>	An unsigned integer in the range 0–1.	1 bit	0
<b>byte</b>	An unsigned integer in the range 0–255.	8 bits	0
<b>time</b>	An integer in the range 0–( $2^{63}-1$ ).	64 bits	0
<b>bool</b>	Represents truth (logical) values, TRUE (1), and FALSE (0).	1 bit	FALSE (0)
<b>real</b>	Represents double precision floating point numbers, identical to the precision of a C type <b>double</b> .	64 bits	0

*Range modifiers* define the range of values that are valid. For example, the range modifier in the following expression restricts valid values to those between 0 and 100, inclusive.

```
int [0..100]
```

*Width modifiers* define the width in bits or bytes. For example, the width modifiers in the following expressions restrict the bit width to 8.

```
int (bits:8);
int (bytes:1)
```

Width and range modifiers can also be used in combination, e.g.,

```
int [0..100] (bits: 7)
```

### 5.1.2.2 Named scalar subtypes

Named scalar subtypes are useful in a context where it is desirable to declare a counter variable, such as the variable `count`, in several places in the program, e.g.,

```
var count : int [0..100] (bits:7);
```

The **type** name can then be used to introduce new variables with this type, e.g.,

```
type int_count : int [0..99] (bits:7);
var count      : int_count
```

See also [5.7.1](#).

### 5.1.2.3 Unbounded integers

Unbounded integers represent arbitrarily large positive or negative numbers. Unbounded integers are specified as:

```
int (bits:*)
```



Use an unbounded integer variable when the exact size of the data is unknown. Unbounded integers can be used in expressions just as signed or unsigned integers are, with the following exceptions:

- Fields or variables declared as unbounded integers shall not be generated, packed, or unpacked.
- Unbounded unsigned integers are not allowed, so a declaration of a type such as `uint (bits:*)` shall generate a compile-time error.

### 5.1.3 Enumerated scalar types

The valid values for a variable or field can be defined as a list of symbolic constants, e.g., the following declaration defines the variable `kind` as having two legal values:

```
var kind : [immediate, register]
```

These symbolic constants have associated unsigned integer values. By default, the first name in the list is assigned the value zero (0). Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items +1. Explicit unsigned integer values can also be assigned to the symbolic constants.

```
var kind : [immediate = 1, register = 2]
```

The associated unsigned integer value of a symbolic constant in an enumerated type can be obtained by using the `as_a()` type casting operator (see [5.8.1](#)). Similarly, an unsigned integer value that is within the range of the values of the symbolic constants can be cast as the corresponding symbolic constant.

Value assignments can also be mixed; some can explicitly be assigned to symbolic constants and others can be automatically assigned. The following declaration assigns the value 3 to `immediate`; the value 4 is automatically assigned to `register`.

```
var kind : [immediate = 3, register]
```

NOTE—Explicitly assigning values to all enumerators aids in avoiding unexpected values.

An enumerated type can be named to facilitate its reuse throughout a program. In the following example, the first statement defines a new enumerated type named `instr_kind`. The variable `i_kind` has the two legal values defined by the `instr_kind` type.

```
type instr_kind : [immediate, register];
var i_kind : instr_kind
```

Enumerated types can also be sized.

```
type instr_kind : [immediate, register] (bits: 2)
```

Variables or fields with an enumerated type can also be restricted to a range. The following variable declaration excludes `foreign` from its legal values:

```
type packet_protocol : [Ethernet, IEEE, foreign];
var p : packet_protocol [Ethernet..IEEE]
```

The default value for an enumerated type is zero (0), even if zero (0) is not a legal value for that type. For example, the variable `i_kind` has the value zero (0) until it is explicitly initialized or generated.

```
type instr_kind : [immediate = 1, register = 2];
var i_kind : instr_kind
```

### 5.1.4 Casting of enumerated types in comparisons

Enumerated scalar types, like Boolean types, are not automatically converted to or from integers or unsigned integers in comparison operations (i.e., comparisons using the `<`, `<=`, `>`, `>=`, `==`, or `!=` operators). This is consistent with the strong typing in *e* and helps avoid the introduction of bugs if the order of symbolic names in an enumerated type declaration is changed. To perform such comparisons, explicit casting or tick notation (`'`) needs to be used to specify the type.

### 5.1.5 Struct types

*Structs* are the basis for constructing compound data structures (see also [Clause 6](#)). The default value for a struct is `NULL`. A struct type can also be used to define a variable (**var**). For more information on **vars**, see [19.2](#).

The following statement creates a struct type called `packet` with a field `protocol` of type `packet_protocol`.

```
struct packet {
    protocol : packet_protocol
}
```

The struct type `packet` can then be used in any context where a type is required. For example, in this statement, `packet` defines the type of a field in another struct.

```
struct port {
    data_in : packet
}
```

### 5.1.6 Struct subtypes

When a struct field has a Boolean type or an enumerated type, a struct subtype can be defined for one or more of the possible values for that field.

#### *Example*

The struct `packet` defined as follows has three possible subtypes based on its `protocol` field. The `gen_eth_packet` method generates an instance of the legal `Ethernet packet` subtype, where `legal == TRUE` and `protocol == Ethernet`.

```
type packet_protocol : [Ethernet, IEEE, foreign];

struct packet {
    protocol : packet_protocol;
    size      : int [0..1k];
    data[size] : list of byte;
    legal      : bool
};

extend sys {
    gen_eth_packet () is {
        var packet : legal Ethernet packet;
        gen packet keeping {it.size < 10};
        print packet
    }
}
```

To refer to a Boolean struct subtype, in this case, `legal packet`, use this syntax:

```
field_name struct_type
```

To refer to an enumerated struct subtype in a struct where no values are shared between the enumerated types, use this syntax:

```
value_name struct_type
```

In structs where more than one enumerated field can have the same value, use the following syntax instead to refer to the struct subtype:

```
value'field_name struct_type
```

The **extend**, **when**, or **like** constructs can also be used to add fields, methods, or method extensions that are required for a particular subtype. Use the **when** or **extend** construct (see [Clause 6](#)) to define struct subtypes with very similar results. These constructs are appropriate for most modeling purposes (see also [Annex C](#)).

### 5.1.7 Referencing fields in when constructs

To refer to a field of a struct subtype outside of a **when**, **like**, or **extend** construct, assign a temporary name to the struct subtype and then use that name. To reference a field in a **when** construct, first specify the appropriate value for the **when** determinant (see [Annex C](#)).

### 5.1.8 List types

*List types* hold ordered collections of data elements, where each data element conforms to the same type. Items in a list can be indexed with the subscript operator [ ], by placing a non-negative integer expression in the brackets. List indexes start at zero (0). To select an item from a list, specify its index, e.g., `my_list[0]` refers to the first item in the list named `my_list`.

Lists are defined by using the **list of** keyword in a variable or a field definition. The following example defines a list of bytes named `lob` and explicitly assigns five literal values to it. The print statement displays the first three elements of `lob`: 15, 31, and 63.

```
var lob : list of byte = {15; 31; 63; 127; 255};
print lob[0..2]
```

The following considerations also apply:

- Multi-dimensional lists (lists of lists) are not supported. To create a list with sublists in it, first create a struct to contain the sublists and then create a list of such structs as the main list.
- The default value of a list is an empty list.
- To set a size for lists that have variable size, use a **keep** constraint or the **resize()** list pseudo-method.

### 5.1.9 Keyed lists

A *keyed list data type* is similar to hash tables or association lists found in other programming languages. If the element type of the list is a scalar type or a string type, then the hash key shall be the predefined implicit variable **it**. The only restriction on the type of the list elements is they themselves shall not be lists. However, they can be struct types containing fields that are lists.

See also [20.4.2](#) and [Clause 27](#).

Syntax example:

```
struct location {
  address : uint;
```

```

    data      : uint
};

struct holder {
    !locations : list(key:address) of location
}

```

### 5.1.10 The string type

The predefined type **string** is the same as the C NULL terminated (zero-terminated) string type. A series of ASCII characters enclosed by quotes (" ") can be assigned to a variable or field of type string, for example:

```

var message : string;
message = "Beginning initialization sequence..."

```

Bits or bit ranges of a string cannot be accessed, but the string can be converted to a list of bytes and that list can be used to access a portion of the string, e.g., the following print statement displays /test1:

```

var dir : string = "/tmp/test1";
var tmp := dir.as_a(list of byte);

tmp = tmp[4..9];
print tmp.as_a(string)

```

The default value of a variable of type **string** is NULL.

See also [20.4.4](#) and [Clause 29](#).

### 5.1.11 The real type

The **real** type in *e* is used to handle and manipulate non-integer numeric values. Real values are physically represented as double-precision floating-point numbers, equivalent to the representation of **double** values in C.

See Section [5.4](#).

### 5.1.12 The external\_pointer type

The **external\_pointer** type is used to hold a pointer into an external (non-*e*) entity, such as a C struct. Unlike pointers to structs in *e*, external pointers are not changed during garbage collection.

Syntax example:

```

var c_handle : external_pointer    // holds a foreign pointer

```

### 5.1.13 The ‘untyped’ pseudo type

This is a type placeholder for untyped values which can be used when runtime values of different types need to be manipulated in a generic way. For example, when objects are manipulated with the reflection API, their types are typically unknown at compile-time; thus, untyped expressions need to be used (see [31.4](#)). Values of any type may be assigned to variables of the ‘untyped’ pseudo type using the **unsafe()** operator (see [5.8.2](#)). Similarly, ‘untyped’ expressions may be used in typed contexts by using **unsafe()**.

NOTE—Untyped variables are left unchanged during garbage collection, which allows struct references to be corrupted.

## 5.2 Untyped expressions

All *e* expressions have an explicit type, except for the following types:

- HDL objects, such as `top.w_en`
- **pack()** expressions, such as `pack(packing.low, 5)`
- bit concatenations, such as `%{s1b1, s1b2}`

The default type of HDL objects is a 32-bit uint, while **pack()** expressions and bit concatenations have a default type of list of bit. However, due to implicit packing and unpacking, these expressions can be converted to the required data type and bit-size in certain contexts, as follows:

- a) When an untyped expression is assigned to a scalar or list of scalars, it is implicitly unpacked and converted to the same type and bit-size as the expression on the LHS. Implicit unpacking is not supported for strings, structs, or lists of non-scalar types.
- b) When a scalar or list of scalars is assigned to an untyped expression, it is implicitly packed before it is assigned. Implicit packing is not supported for strings, structs, or lists of non-scalar types.
- c) When the untyped expression is the operand of any binary operator (+, −, \*, /, or %), the expression is assumed to be a numeric type. The precision of the operation is determined by the expected type and the type of the operands (see [5.5](#)).
- d) When a **pack()** expression includes the parameter or the return value of a method call, the expression takes the type and size as specified in the method declaration. The method parameter or return value in the pack expression shall be a scalar type or a list of scalar type.
- e) When an untyped expression appears in one of the following contexts, it is treated as a Boolean expression:

```
if (untyped_exp) then {..}
while (untyped_exp) do {..}
check that (untyped_exp)
not untyped_exp
rise(untyped_exp), fall(untyped_exp), true(untyped_exp)
```

When the type and bit-size cannot be determined from the context, the expression is automatically cast according to the following rules:

- The default type of an HDL signal is an unsigned integer; the default bit-size is 32.
- The default type of a pack expression and a bit concatenation expression is a list of bit.

When expressions are untyped, an implicit pack/unpack is performed according to the expected type. See also [20.5](#).

## 5.3 Assignment rules

Assignment rules define what is a legal assignment and how values are assigned to entities. The following subclauses describe various aspects of assignments.

### 5.3.1 What is an assignment?

There are several legal ways to assign values:

- Assignment actions
- Return actions
- Parameter passing

### — Variable declaration

Here is an example of an assignment action, where a value is explicitly assigned to a variable `x` and to a field `sys.x`.

```
extend sys {
  x : int;
  m() is {
    var x: int;
    sys.x = '~/top/address';
    x = sys.x + 1
  }
}
```

Here is an example of a **return** action, which implicitly assigns a value to the **result** variable:

```
extend sys {
  n(): int (bits:64) is {
    return 1
  }
}
```

Here is an example of assigning a value (6) to a method parameter (`i`):

```
extend sys {
  k(i: int) @sys.any is {
    wait [i] * cycle
  };

  run() is also {
    start k(6)
  }
}
```

Here is an example of how variables are assigned during declaration:

```
extend sys {
  b() is {
    var x : int = 5;
    var y := "ABC"
  }
}
```

Values shall not be assigned to fields during declaration, however.

### 5.3.2 Assignments create identical references

Assigning one struct, list, or value to another object of the same type results in two references pointing to the same memory location, so that changes to one of the objects also occur in the other object immediately.

#### *Example*

```
data1 : list of byte;
data2 : list of byte;

run() is also {
  data2 = data1;
  data1[0] = 0
}
```

After generation, the two lists `data1` and `data2` are different lists. However, after the `data2 = data1` assignment, both lists refer to the same memory location; therefore, changing the `data1[0]` value also changes the `data2[0]` value immediately.

### 5.3.3 Assignment to different (but compatible) types

This subclause describes the assignment between different, yet compatible, types.

#### 5.3.3.1 Assignment of numeric types

Any numeric type (e.g., **uint**, **int**, or one of their subtypes) can be assigned with any other numeric type. Untyped expressions, such as HDL objects, can also appear in assignments of numeric types (see [5.2](#)).

Automatic casting is performed when a numeric type is assigned to a different numeric type, and automatic extension or truncation is performed if the types have a different bit-size (see [5.6](#)). See also [5.5](#).

#### 5.3.3.2 Assignment of Boolean types

A Boolean type can only be assigned to another Boolean type.

```
var x : bool;
x = 'top.a' >= 16
```

#### 5.3.3.3 Assignment of enumerated types

An enumerated type can be assigned with that same type or its scalar subtype. (The scalar subtype differs only in range or bit-size from the base type.) The following example shows:

- An assignment of the same type

```
var x : color = blue
```

- An assignment of a scalar subtype

```
var y : color2 = x
```

To assign any scalar type (numeric, enumerated, or Boolean type) to any different scalar type, use the **as\_a()** operator (see [5.8.1](#)).

#### 5.3.3.4 Assignment of structs

An entity of type struct can be assigned with a struct of that same type or with one of its subtypes. The following example shows:

- A same type assignment

```
p2 = p1
```

- An assignment of a subtype (`Ether_8023 packet`)

```
var p : Ether_8023 packet;
set_cell(p)
```

- An assignment of a derived struct (`cell_8023`)

```
set_cell(p:packet) is {
  p.cell = new cell_8023;
  ...
}
```

Although a subtype can be assigned to its parent struct without any explicit casting, to perform the reverse assignment (assign a parent struct to one of its subtypes), the `as_a()` method needs to be used (see [5.8.1](#)).

### 5.3.3.5 Assignment of strings

A string can be assigned only with strings, as follows:

```
extend sys {
  m(): string is {
    return "aaa"    // assignment of a string
  }
}
```

### 5.3.3.6 Assignment of lists

An entity of a type list can be assigned only with a list of the same type. In the following example, the assignment of `list1` to `x` is legal because both lists are lists of integers:

```
extend sys {
  list1 : list of int;
  m() is {
    var x : list of int = list1;
  }
}
```

However, an assignment such as `var y: list of int (bits: 16) = list1;` is not legal, because `list1` is not the same list type as `y`. `y` has a size modifier, so it is a subtype of `list1`.

Use the `as_a()` method to cast between lists and their subtypes (see [5.8.1](#)).

## 5.4 Real data type

Objects of type **real** are double precision floating point numbers, the same as C type **double**. The representation of real values and the semantics of arithmetic and cast operators uses the double-precision floating point implementation on the underlying machine, which should be compliant with the IEEE754 standard.

### 5.4.1 Real data type usage

In any context expecting a numeric object, a **real** object is acceptable except in the following cases:

- Both operands of the shift operators (<<, >>)
- Bitwise operators (|, &, ^)
- Bitwise routines
- Modulo (%)
- `odd()`
- `even()`

### 5.4.2 Real literals

**Real** literals are numbers that have a decimal point or an exponential part or both. If a decimal point exists, there must be digits on both sides of the decimal point. Underscores can be added for readability and are ignored.



**Table 18—Examples of real literals**

Real Constant	Value
5.3876e4	53,876
4e-11	0.00000000004
1e+5	100,000
7.321E-3	0.007321
3.2E+4	32,000
0.5e-6	0.0000005
0.45	0.45
6.e10	60,000,000,000

### 5.4.3 Real constants

The **real** constants in Table 19 and Table 20 are defined in both *e* code and in C code that includes a suitable header file:

**Table 19—Mathematical constants**

Constant	Value
IEEE_1647_M_E	e
IEEE_1647_M_LOG2E	Logarithm base 2 of e
IEEE_1647_M_LOG10E	Logarithm base 10 of e
IEEE_1647_M_LN2	Natural logarithm of 2
IEEE_1647_M_LN10	Natural logarithm of 10
IEEE_1647_M_PI	PI
IEEE_1647_M_TWO_PI	2*PI
IEEE_1647_M_PI_2	PI/2
IEEE_1647_M_PI_4	PI/4
IEEE_1647_M_1_PI	1/PI
IEEE_1647_M_2_PI	2/PI
IEEE_1647_M_2_SQRTPI	2/sqrt(PI)
IEEE_1647_M_SQRT2	sqrt(2)
IEEE_1647_M_SQRT1_2	sqrt(1/2)

NOTE— All mathematical constants are prefixed by IEEE\_1647\_M\_.

NOTE— All physical constants are prefixed by IEEE\_1647\_P\_.

**Table 20—Physical constants**

Constant	Value
IEEE_1647_P_Q	Charge of electron in coulombs
IEEE_1647_P_C	Speed of light in vacuum in meters/sec
IEEE_1647_P_K	Boltzmann's constant in joules/kelvin
IEEE_1647_P_H	Planck's constant in joules*sec
IEEE_1647_P_EPS0	Permittivity of vacuum in farads/meter
IEEE_1647_P_U0	Permeability of vacuum in henrys/meter
IEEE_1647_P_CELSIUS0	Zero Celsius in kelvin

#### 5.4.4 Real type limitations

- The key of a keyed list cannot be of type **real**.

### 5.5 Precision rules for numeric operations

For precision rules, there are two types of numeric expressions in *e*, as follows:

- *context-independent* expressions, where the precision of the operation (bit width) and numeric type (signed or unsigned) depend only on the types of the operands
- *context-dependent* expressions, where the precision of the operation and the numeric type depend on the precision and numeric type of other expressions involved in the operation (the *context*), as well as the types of the operands

A numeric operation in *e* is performed in one of three possible combinations of precision and numeric type:

- Unsigned 32-bit integer (**uint**)
- Signed 32-bit integer (**int**)
- Infinite signed integer (**int (bits: \*)**)

The *e* language has rules for determining the context of an expression or deciding the precision, and performing data conversion and sign extension.

#### 5.5.1 Determining the context of an expression

The rules for defining the context of an expression are applied in the following order:

- In an assignment (*lhs = rhs*), the right-hand side (*rhs*) expression inherits the context of the left-hand side (*lhs*) expression.
- A sub-expression inherits the context of its enclosing expression.
- In a binary-operator expression (*lho OP rho*), the right-hand operand (*rho*) inherits context from the left-hand operand (*lho*), as well as from the enclosing expression.

[Table 21](#) summarizes context inheritance for each type of operator that can be used in numeric expressions.

**Table 21—Summary of context inheritance in numeric operations**

Operator	Function	Context
<code>* / % + -</code> <code>&lt; &lt;= &gt; &gt;=</code> <code>== !=</code> <code>=== !==</code> <code>&amp;   ^</code>	Arithmetic, comparison, equality, and bit-wise Boolean	The right-hand operand ( <i>rho</i> ) inherits context from the left-hand operand ( <i>lho</i> ), as well as from the enclosing expression. <i>lho</i> inherits only from the enclosing expression.
<code>~ !</code> unary <code>+</code> <code>-</code>	Bit-wise not, Boolean not, unary plus, minus	The operand inherits context from the enclosing expression.
<code>[ ]</code>	List indexing	The list index is context-independent.
<code>[ .. ]</code>	List slicing	The indices of the slice are context-independent.
<code>[ : ]</code>	Bit slicing	The indices of the slice are context-independent.
<code>f(...)</code>	Method or routine call	The context of a parameter to a method is the type and bit width of the formal parameter.
<code>{...; ...}</code>	List concatenation	Context is passed from the <i>lhs</i> of the assignment, but not from left-to-right between the list members.
<code>%{..., ...}</code>	Bit concatenation	The elements of the concatenation are context-independent.
<code>&gt;&gt;, &lt;&lt;</code>	Shift	Context is passed from the enclosing expression to the left operand. The context of the right operand is always a 32-bit <b>uint</b> .
<code>lho in [i..j]</code>	Range list operator	All three operands are context-independent. (The range specifiers <i>i</i> and <i>j</i> shall be constant.)
<code>&amp;&amp;,   </code>	Boolean	All operands are context-independent.
<code>a ? b : c</code>	Conditional operator	<i>a</i> is context-independent, <i>b</i> inherits the context from the enclosing expression, <i>c</i> inherits context from <i>b</i> , as well as from the enclosing expression.
<code>as_a()</code>	Casting	The operand is context-independent.
<code>abs()</code> , <code>odd()</code> , <code>even()</code>	Arithmetic routine	The parameter is context-independent.
<code>min()</code> , <code>max()</code>	Arithmetic routine	The right parameter inherits context from the left parameter ( <i>lp</i> ), as well as from the enclosing expression. <i>lp</i> inherits only from the enclosing expression.
<code>ilog2()</code> , <code>ilog10()</code> , <code>isqrt()</code>	Arithmetic routine	The context of the parameter is always a 32-bit <b>uint</b> .
<code>ipow()</code>	Arithmetic routine	Both parameters inherit the context of the enclosing expression, but the right parameter does not inherit context from the left.

### 5.5.2 Deciding precision and performing data conversion and sign extension

The rules for deciding precision, and performing data conversion and sign extension are as follows:

Determine the context of the expression, which can be comprised of a maximum of two types:

- a) If all types involved in an expression and its context are 32 bits in width or less:
  - 1) The operation is performed in 32 bits.
  - 2) If any of the types is unsigned, the operation is performed with unsigned integers.

Decimal constants are treated as signed integers, whether they are negative or not. All other constants are treated as unsigned integers, unless preceded by a hyphen (-).

- 3) Each operand is automatically cast, if necessary, to the required type.

Casting of small negative numbers (signed integers) to unsigned integers produces large positive numbers.

- b) If any of the types is greater than 32 bits:
  - 1) The operation is performed in infinite precision [**int (bits:\*)**].
  - 2) Each operand is zero-extended (if it is unsigned) or sign-extended (if it is signed) to infinite precision.

## 5.6 Automatic type casting

During assignment of a type to a different but compatible type, automatic type casting is performed in the following contexts:

- Numeric expressions (unsigned and signed integers) of any size are automatically type cast upon assignment to different numeric types. For example:
 

```
var x : uint;
var y : int;
x = y
```
- Untyped expressions are automatically cast on assignment. See [5.2](#) for more information.
- Sized scalars are automatically type cast to differently sized scalars of the same type.
- Struct subtypes are automatically cast to their base struct type.

There are three important ramifications to automatic type casting.

- a) If the two types differ in bit-size, the assigned value is extended or truncated to the required bit-size.
- b) Casting of small negative numbers (signed integers) to unsigned integers produces large positive numbers.
- c) There is no automatic casting to a reference parameter (see [18.3](#)).

### 5.6.1 Conversion between real and integer data types

Automatic casting is performed between the **real** type and the other numeric types.

Converting a **real** type object to an integer type object uses the following process:

- a) The object is first converted to type **int (bits:\*)** with the value of the largest integer whose absolute value is less than or equal to the absolute value of the **real** object.
- b) The object is then converted to the expected integer type.

Additional rules apply to converting **real** objects to integer objects:

- If the object's floating-point value is infinity (inf), negative infinity (-inf), or Not-a-Number (NaN), an error will be emitted when trying to convert to an integer value.
- When converting an integer object to the **real** type, the object is converted to the value closest to the integer value that can be represented in the double precision format.

When converting from an integer data type to a **real**, the integer value is simply converted to its identical value represented as a real.

Automatic casting of reals to integers or integers to reals is not performed in the context of constraints. Explicit casting is required within constraints that involve both integer and real expressions so that all resulting terms are of the same kind.

### 5.6.2 Real data type precision, data conversion, and sign extension

The rules for deciding precision, performing data conversion, and sign extension are as follows:

- a) Determine the context of the expression. The context may be comprised of up to three types.
- b) If all types involved in an expression and its context are integer values of 32 bits in width or less:
  - 1) The operation is performed in 32 bits.
  - 2) If any of the types are unsigned, the operation is performed with unsigned integers.  
NOTE—Decimal constants are treated as signed integers, whether they are negative or not. All other constants are treated as unsigned integers unless preceded by a hyphen.
  - 3) Each operand is automatically cast, if necessary, to the required type.  
NOTE—Casting of small negative numbers (signed integers) to unsigned integers produces large positive numbers.
- c) If all types are integer types, and any of the types is greater than 32 bits:
  - 1) The operation is performed in infinite precision (int(bits:\*)).
  - 2) Each operand is zero-extended if it is unsigned, or sign-extended if it is signed, to infinite precision.
- d) If any of the types is a **real** type, then the operation is done in double precision, and all objects should first be converted according to the rules described above.

## 5.7 Defining and extending scalar types

The following constructs can be used to define and extend scalar types.

### 5.7.1 type enumerated scalar

<b>Purpose</b>	Define an enumerated scalar type
<b>Category</b>	Statement
<b>Syntax</b>	<b>type</b> <i>enum-type-name</i> : <b>[</b> <i>[name</i> <b>[</b> <i>=exp</i> <b>], ...</b> <b>]</b> <b>[</b> <b>(bits   bytes:</b> <i>width-exp</i> <b>)</b> <b>]</b>
<b>Parameters</b>	<i>enum-type-name</i> A legal <i>e</i> name. The name shall be unique in the global type-name space.
	<i>name</i> A legal <i>e</i> name. Each name shall be unique within the same type.
	<i>exp</i> A unique 32-bit constant expression. Names or name-value pairs can appear in any order. By default, the first name in the list is assigned the integer value zero (0). Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items + 1.
	<i>width-exp</i> A positive constant expression. The valid range of values for sized enumerated scalar types is limited to the range 1 to $2^{n-1}$ , where <i>n</i> is the number of bits.

This defines an enumerated scalar type consisting of a set of names or name-value pairs. If no values are specified, the names get corresponding numerical values starting with 0 for the first name, and casting can be done between the names and the numerical values.

Syntax example:

```
type PacketType : [rx=1, tx, ctrl]
```

### 5.7.2 type scalar subtype

<b>Purpose</b>	Define a scalar subtype	
<b>Category</b>	Statement	
<b>Syntax</b>	<b>type</b> <i>scalar-subtype-name</i> : <i>scalar-type</i> [ <i>range</i> , ...]	
<b>Parameters</b>	<i>scalar-subtype-name</i>	A unique <i>e</i> name. The name shall be unique in the global type-name space.
	<i>scalar-type</i>	Any previously defined enumerated scalar type, any of the predefined scalar types, including <b>int</b> , <b>uint</b> , <b>bool</b> , <b>bit</b> , <b>byte</b> , or <b>time</b> , or any previously defined scalar subtype.
	<i>range</i>	A constant expression or two constant expressions separated by two dots ( . . ). All constant expressions shall resolve to legal values of the named type.

This defines a subtype of a scalar type by restricting the legal values that can be generated for this subtype to the specified range. The default value for variables or fields of this type “size” is zero (0), which is the default for all integers. The *range* only affects any generated values.

Syntax example:

```
type size : int [8, 16]
```

### 5.7.3 type sized scalar

<b>Purpose</b>	Define a sized scalar	
<b>Category</b>	Statement	
<b>Syntax</b>	<b>type</b> <i>sized-scalar-name</i> : <i>type</i> ( <b>bits</b>   <b>bytes</b> : <i>exp</i> )	
<b>Parameters</b>	<i>sized-scalar-name</i>	A unique <i>e</i> name. The name shall be unique in the global type-name space.
	<i>type</i>	Any previously defined enumerated type or any of the predefined scalar types, including <b>int</b> , <b>uint</b> , <b>bool</b> , or <b>time</b> .
	<i>exp</i>	A positive constant expression. The valid range of values for sized scalars is limited to the range 1 to $2^{n-1}$ , where $n$ is the number of bits.

This defines a scalar type with a specified bit width. The actual bit width is  $exp * 1$  for bits and  $exp * 8$  for bytes.

When assigning any expression into a sized scalar variable or field, the expression’s value is truncated or extended automatically to fit into the variable. An expression with more bits than the variable is chopped down to the size of the variable. An expression with fewer bits is extended to the length of the variable. The added upper bits are filled with zeros (0) if the expression is unsigned or with the appropriate sign bit (0 or 1) if the expression is signed.

Syntax example:

```
type word      : uint( bits:16);
type address  : uint(bytes: 2)
```

#### 5.7.4 extend type

<b>Purpose</b>	Extend an enumerated scalar type
<b>Category</b>	Statement
<b>Syntax</b>	<b>extend</b> <i>enum-type</i> : [ <i>name</i> [= <i>exp</i> ], ...]
<b>Parameters</b>	<i>enum-type</i> Any previously defined enumerated type.
	<i>name</i> A legal <i>e</i> name. Each name shall be unique within the type.
	<i>exp</i> A unique 32-bit constant expression. Names or name-value pairs can appear in any order. By default, the first name in the list is assigned the integer value zero (0). Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items +1.

This extends the specified enumerated scalar type to include the specified names or name-value pairs.

Syntax example:

```
type PacketType : [rx, tx, ctrl];
extend PacketType : [status]
```

### 5.8 Type-related constructs

The **as\_a()** expression is used to convert an expression from one data type to another. The **unsafe()** expression casts the expression to the type that is required by the context. The **all\_values()** pseudo-routine returns a list of all of the legal values of a specified scalar type. Information about how different types are converted, such as strings to scalars or lists of scalars, is contained in [Table 22](#) and [Table 23](#).

#### 5.8.1 as\_a()

<b>Purpose</b>	Casting operator
<b>Category</b>	Expression
<b>Syntax</b>	<i>exp</i> . <b>as_a</b> ( <i>type</i> : type name): type
<b>Parameters</b>	<i>exp</i> Any <i>e</i> expression.
	<i>type</i> Any legal <i>e</i> type.

This returns the expression converted into the specified type. Although some casting is done automatically (see [5.6](#)), explicit casting is required to make assignments between different but compatible types.

Following are assignment compatible types requiring explicit casting:

- Scalars and lists of scalars

- Strings and scalars or lists of scalars
- Structs and list of structs
- Simple lists and keyed lists

Syntax example:

```
print (b).as_a(uint)
```

### 5.8.2 unsafe()

<b>Purpose</b>	Force casting
<b>Category</b>	Expression
<b>Syntax</b>	<i>exp</i> .unsafe()
<b>Parameters</b>	<i>exp</i> Any <i>e</i> expression.

This casts the expression to the type that is required by the context, regardless of any static or dynamic type rules. This operator may be used only in contexts where the required type is explicit, such as assignment and parameter passing to methods.

Syntax example:

```
var value : int = param.unsafe()
```

#### 5.8.2.1 Type conversion between scalars and lists of scalars

Numeric expressions (unsigned and signed integers) of any size are automatically type cast upon assignment to different numeric types.

For other scalars and lists of scalars, there are a number of ways to perform type conversion, including the **as\_a()** method, the **pack()** method, the **%{}** bit concatenation operator, and various string routines. [Table 22](#) shows how to convert between scalars and lists of scalars.

In [Table 22](#), **int** represents **int/uint** of any size, including bit, byte, and any user-created size. If a solution is specific to bit or byte, then bit or byte is explicitly stated. **int(bits:x)** means *x* as any constant; variables shall not be used as the integer width.

The solutions presume variables are declared as

```
var int          : int;
var bool         : bool;
var enum         : enum;
var list_of_bit  : list of bit;
var list_of_byte : list of byte;
var list_of_int  : list of int
```

Any conversions not explicitly shown might have to be accomplished in two stages.



**Table 22—Type conversion between scalars and lists of scalars**

From	To	Solutions
<i>int</i>	<i>list of bit</i>	<code>list_of_bit = int[..]</code>
<i>int</i>	<i>list of int(bits:x)</i>	<code>list_of_int = %{int}</code> <code>list_of_int = pack(packing.low, int)</code> (LSB of int goes to list[0] for either choice)
<i>list of bit list of byte</i>	<i>int</i>	<code>int = list_of_bit[:]</code>
<i>list of int(bits:x)</i>	<i>int</i>	<code>int = pack(packing.low, list_of_int)</code> (use <code>packing.high</code> for list in the other order)
<i>int(bits:x)</i>	<i>int(bits:y)</i>	<code>intx = inty</code> (truncation or extension is automatic) <code>intx.as_a(int(bits:y))</code>
<i>bool</i>	<i>int</i>	<code>int = bool.as_a(int)</code> (TRUE becomes 1, FALSE becomes 0)
<i>int</i>	<i>bool</i>	<code>bool = int.as_a(bool)</code> (0 becomes FALSE, non-0 becomes TRUE)
<i>int</i>	<i>enum</i>	<code>enum = int.as_a(enum)</code> (no checking is performed to make sure the int value is valid for the range of the enum)
<i>enum</i>	<i>int</i>	<code>int = enum.as_a(int)</code> (truncation is automatic)
<i>enum</i>	<i>bool</i>	<code>enum.as_a(bool)</code> [enumerated types with an associated unsigned integer value of 0 become FALSE; those with an associated non-0 values become TRUE (see 5.1.3)]
<i>bool</i>	<i>enum</i>	<code>bool.as_a(enum)</code> (Boolean types with a value of FALSE are converted to the enumerated type value that is associated with the unsigned integer value of 0; those with a value of TRUE are converted to the enumerated type value that is associated with the unsigned integer value of 1; no checking is performed to make sure the Boolean value is valid for the range of the enum)
<i>enum</i>	<i>enum</i>	<code>enum1 = enum2.as_a(enum1)</code> (no checking is performed to make sure the int value is valid for the range of the enum)
<i>list of int(bits:x)</i>	<i>list of int(bits:y)</i>	<code>listx.as_a(list of int(bits:y))</code> (the same number of items, each padded or truncated) <code>listy = pack(packing.low, listx)</code> (concatenated data, different number of items)

### 5.8.2.2 Type conversion between strings and scalars or lists of scalars

There are a number of ways to perform type conversion between strings and scalars or lists of scalars, including the `as_a()` method, the `pack()` method, the `%{}` bit concatenation operator, and various string routines. Table 23 shows how to convert between strings and scalars or lists of scalars.

In Table 23, **int** represents **int/uint** of any size, including bit, byte, and any user-created size. If a solution is specific to bit or byte, then bit or byte is explicitly stated. **int(bits:x)** means *x* as any constant; variables shall not be used as the integer width.

**Table 23—Type conversion between strings and scalars or lists of scalars**

From	To	ASCII convert?	Solutions
<i>list of int</i> <i>list of byte</i>	<i>string</i>	Yes	<code>list_of_int.as_a(string)</code> (each list item is converted to its ASCII character and the characters are concatenated into a single string; <code>int[0]</code> represents left-most character; if a list item is not a printable ASCII character, the string returned is empty)
<i>string</i>	<i>list of int</i> <i>list of byte</i>	Yes	<code>string.as_a(list of int)</code> (each character in the string is converted to its numeric value and assigned to a separate element in the list; the left-most character becomes <code>int[0]</code> )
<i>string</i>	<i>list of int</i>	Yes	<code>list_of_int = pack(packing.low, string)</code> <code>list_of_int = %{string}</code> (the numeric values of the characters are concatenated before assigning them to the list; any pack option gives same result; the null byte, 00, is the last item in the list)
<i>string</i>	<i>int</i>	Yes	<code>int = %{string}</code> <code>int = pack(packing.low, string)</code> (any pack option gives the same result)
<i>int</i>	<i>string</i>	Yes	<code>unpack(packing.low, %{8'b0, int}, string)</code> (any pack option with <code>scalar_reorder={ }</code> gives the same result)
<i>string</i>	<i>int</i>	No	<code>string.as_a(int)</code> (converts to decimal) <code>append("0b", string).as_a(int)</code> (converts to binary) <code>append("0x", string).as_a(int)</code> (converts to hexadecimal)
<i>int</i>	<i>string</i>	No	<code>int.as_a(string)</code> (uses the current print radix) <code>append(int)</code> (converts int according to the current print radix) <code>dec(int)</code> , <code>hex(int)</code> , <code>bin(int)</code> (converts int according to a specific radix)
<i>string</i>	<i>bool</i>	No	<code>bool = string.as_a(bool)</code> (only TRUE and FALSE can be converted to Boolean; all other strings return an error)
<i>bool</i>	<i>string</i>	No	<code>string = bool.as_a(string)</code>
<i>string</i>	<i>enum</i>	No	<code>enum = string.as_a(enum)</code>
<i>enum</i>	<i>string</i>	No	<code>string = enum.as_a(string)</code>

The solutions presume variables are declared as follows:

```

var int           : int;
var list_of_byte  : list of byte;
var list_of_int   : list of int;
var bool          : bool;
var enum          : enum;
var string        : string

```

Any conversions not explicitly shown might have to be accomplished in two stages.

### 5.8.2.3 Type conversion between structs, struct subtypes, and lists of structs

Struct subtypes are automatically cast to their base struct type, so for example, a variable of type Ethernet packet can be assigned to a variable of type packet without using `as_a()`. `as_a()` can be used to cast a base struct type to one of its subtypes; if a mismatch occurs, then NULL is assigned. For example, the `print pkt.as_a(foreign packet)` action results in `pkt.as_a(foreign packet) = NULL` if `pkt` is not a foreign packet.

When the expression to be converted is a list of structs, `as_a()` returns a new list of items whose type matches the specified type parameter. If no items match the type parameter, an empty list is returned. The list can contain items of various subtypes, but all items shall have a common parent type, i.e., the specified type parameter shall be a subtype of the type of the list.

Assigning a struct subtype to a base struct type does not change the declared type. Thus, `as_a()` needs to be used to cast the base struct type as the subtype and access any of the subtype-specific struct members.

Subtypes created through **like** inheritance exhibit the same behavior as subtypes created through **when** inheritance.

### 5.8.2.4 Type conversion between simple lists and keyed lists

Simple lists can be converted to keyed lists and vice versa. The hash key is dropped in converting a keyed list to a simple list. However, a key needs to be specified first to convert a simple list to a keyed list.

#### *Example*

To convert a simple list of packets `sys.packets` to a keyed list, where the `len` field of the packet struct is the key:

```
var pkts : list (key:len) of packet;
pkts = sys.packets.as_a(list (key:len) of packet)
```

Using the `as_a()` method returns a copy of `sys.packets`, so the original `sys.packets` is still a simple list, not a keyed list. Thus, `print pkts.key_index(130)` returns the index of the item that has a `len` field of 130, while `print sys.packets.key_index(130)` shall return an error.

If a conversion between a simple list and a keyed list also involves a conversion of the type of each item, that conversion of each item follows the standard rules, e.g., when `as_a()` is used to convert an integer to a string, no ASCII conversion is performed. Similarly, if `as_a()` is used to convert a simple list of integers to a keyed list of strings, no ASCII conversion is performed.

No checking is performed to ensure the value is valid when casting from a numeric or Boolean type to an enumerated type, or when casting between enumerated types.

- The `as_a()` pseudo-method, when applied to a scalar list, creates a new list whose size is the same as the original size and then casts each element separately.
- When the `as_a()` operator is applied to a list of structs, the list items for which the casting failed are omitted from the list.
- `as_a()` can be used to convert a string to an enumerated type. The string has to exactly match one of the possible values of that type, using a case-sensitive string comparison, or a runtime error shall be issued.

See also [4.16.1](#).

### 5.8.2.5 Type conversion between reals and non-numeric scalars

Converting a non-numeric scalar type object to a **real** type object using the **as\_a()** operator uses the following process:

- a) The scalar type object is first converted to an integer value.
- b) The object is then converted to a **real** value according to process and rules listed in [5.4.4](#).

Additional rules apply to converting non-numeric scalar objects to **real** objects using the **as\_a()** operator:

- When converting a string value to **real** using the **as\_a()** operator, the string is parsed as if it was a **real** literal, and the value of the **real** literal is returned.
- If the string does not conform to the definition of a **real** literal, an error is emitted.

### 5.8.3 all\_values()

<b>Purpose</b>	Access all values of a scalar type
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>all_values</b> ( <i>scalar-type</i> : type name): list of scalar type
<b>Parameters</b>	<i>scalar-type</i> Any legal <i>e</i> scalar type.

This returns a list that contains all the legal values of the specified scalar type. When that type is an enumerated type, the order of the items is the same as the order in which they were defined. When the type is a numeric type, the order of the items is from the smallest to the largest.

Syntax example:

```
print all_values(reg_address)
```

## 6. Structs, subtypes, and fields

The basic organization of an *e* program is a tree of structs. A struct is a compound type that contains data fields, procedural methods, and other members. It is the *e* equivalent of a class in other OO languages. A base struct type can be extended by adding members. Subtypes can be created from a base struct type, which inherit the base type's members and contain additional members. A *field* is a feature of a struct that can hold data. Fields can be scalars or references to structs or lists.

An extension can reside in a module outside of where it was originally defined, in which case, the extending module shall be loaded after the original module. For more details about load ordering, see [Annex B](#).

### 6.1 Structs overview

Structs are used to define data elements and behavior of components of a test environment.

- A struct can hold all types of data and methods.
- All user-defined structs inherit from the predefined base struct type, **any\_struct**.
- For reusability of *e* code, use **extend** to add struct members or change the behavior of a previously defined struct.

Inheritance is implemented in *e* by using either of the following mechanisms:

- a) “when” inheritance is specified by defining subtypes with **when** struct members.
- b) “like” inheritance is specified with the **like** clause in new struct definitions.

The best inheritance methodology for most applications is “when” inheritance (see [31.2.4](#) and [Annex C](#)). Note that struct types can also be defined implicitly through the instantiation of templates. Template instances can be extended, inherited from, or used as variable types, just like explicitly defined struct types. See [Clause 8](#).

## 6.2 Defining structs: struct

<b>Purpose</b>	Define a data struct
<b>Category</b>	Statement
<b>Syntax</b>	<b>struct</b> <i>struct-type</i> [ <b>like</b> <i>base-struct-type</i> ] { [ <i>struct-member</i> ; ...]}
<b>Parameters</b>	<i>struct-type</i> The name of the new struct type.
	<i>base-struct-type</i> The type of the struct from which the new struct inherits its members.
	<i>struct-member</i> ; ...   The contents of the struct. The following are types of struct members: <ul style="list-style-type: none"> <li>— data fields for storing data</li> <li>— methods for procedures</li> <li>— events for defining temporal triggers</li> <li>— coverage groups for defining coverage points</li> <li>— <b>when</b>, for specifying inheritance subtypes</li> <li>— declarative constraints for describing relations between data fields</li> <li>— <b>on</b>, for specifying actions to perform upon event occurrences</li> <li>— <b>expect</b>, for specifying temporal behavior rules</li> </ul> The definition of a struct can be empty, containing no members.

Structs are used to define the data elements and behavior of components and the test environment. Structs contain struct members of the types listed in the preceding *Parameters* description. Struct members can be conditionally defined (see [6.6](#)).

The optional **like** clause is an inheritance directive. All struct members defined in *base-struct-type* are implicitly defined in the struct subtype, *struct-type*. New struct members can also be added to the inheriting struct subtype, and methods of the base struct type can be extended in the inheriting struct subtype.

Additional subtypes can, in turn, be derived from a subtype. In the following example, the subtype `agp_transaction` is derived from the (previously defined) `pci_transaction` subtype. Each subtype can add fields to its base type and place its own constraints on fields of its base type.

Syntax example:

```

type AGPModeType : [AGP_2X, AGP_4X];

struct agp_transaction like pci_transaction {
    block_size : uint;
    mode       : AGPModeType;

    when AGP_2X agp_transaction {
        keep block_size == 32
    };

    when AGP_4X agp_transaction {
        keep block_size == 64
    }
}

```

### 6.3 Extending structs: extend type

<b>Purpose</b>	Extend an existing data struct
<b>Category</b>	Statement
<b>Syntax</b>	<b>extend</b> [ <i>struct-subtype</i> ] <i>base-struct-type</i> { [ <i>struct-member</i> ; ...]}
<b>Parameters</b>	<i>struct-subtype</i> Adds struct members to the specified subtype of the base struct type only. The added struct members are known only in that subtype, not in other subtypes.
	<i>base-struct-type</i> The base struct type to extend.
	<i>struct-member</i> ; ...    The contents of the struct. The following are types of struct members: <ul style="list-style-type: none"> <li>— data fields for storing data</li> <li>— methods for procedures</li> <li>— events for defining temporal triggers</li> <li>— coverage groups for defining coverage points</li> <li>— <b>when</b>, for specifying inheritance subtypes</li> <li>— declarative constraints for describing relations between data fields</li> <li>— <b>on</b>, for specifying actions to perform upon event occurrences</li> <li>— <b>expect</b>, for specifying temporal behavior rules</li> </ul> The definition of a struct can be empty, containing no members.

This adds struct members to a previously defined struct or struct subtype. Members added to the base struct type in extensions apply to all other extensions of the same struct, e.g., if a method in a base struct is extended using **is only**, it overrides that method in every one of the **like** children.

If **like** inheritance has been used on a struct type, there are limitations on further extending the original base struct type definition; see [6.4](#).

Syntax example:

```

type packet_kind : [atm, eth];

struct packet {
    len : int;
    kind : packet_kind
};

extend packet {
    keep len < 256
}

```

### 6.4 Restrictions on inheritance

The following restrictions shall apply when using inheritance:

- Generation of a parent does not create **like** children.
- **when** subtypes shall not be added to a struct with **like** children. Similarly, a **like** child shall not be created from a struct that has **when** subtypes.

## 6.5 Extending subtypes

A *struct subtype* is an instance of the struct in which one of its fields has a particular value. For example, the `packet` struct defined in the following example has `atm packet` and `eth packet` subtypes, depending on whether the `kind` field is `atm` or `eth`.

*Example*

```
type packet_kind : [atm, eth];

struct packet {
  len  : int;
  kind : packet_kind
};

extend atm packet {
  keep len == 53
}
```

Similar to structs, a struct subtype can be **extended**; the extension shall only apply to that subtype.

## 6.6 Creating subtypes with when

The **when** struct member creates a conditional subtype of the current struct type when a particular field of the struct has a given value. This is called “when” inheritance and is one of two techniques that *e* provides for implementing inheritance. The other is called “like” inheritance. When inheritance is described in this subclause. Like inheritance is described in [6.2](#).

When inheritance is the recommended technique for modeling in *e*. Like inheritance is more appropriate for procedural test bench programming. When and like inheritance are compared in [Annex C](#).



### 6.6.1 when

<b>Purpose</b>	Define a struct subtype
<b>Category</b>	Struct member
<b>Syntax</b>	<b>when</b> <i>determinant</i> ... [ <i>base-struct-type</i> ] { <i>struct-member</i> ; ...}
<b>Parameters</b>	<i>determinant</i> One or more subtype determinants separated by a space. A subtype determinant is in one of the following two forms: <ul style="list-style-type: none"> <li>a) <i>value</i>[<i>'field-name'</i>] - where <i>value</i> is a legal value of one of the context struct's enumerated fields, and <i>field-name</i> is the name of the corresponding field. When <i>field-name</i> is omitted, <i>value</i> must be associated with one of the struct's fields unambiguously.</li> <li>b) [(TRUE FALSE)']<i>field-name</i> - where <i>field-name</i> is the name of a Boolean field of the context struct. If a Boolean value is omitted, it is taken to be TRUE.</li> </ul>
	<i>base-struct-type</i> The name of the context base struct type to which the when determinants apply. Providing this parameter is optional (it is inferred from the syntactic context if omitted).
	<i>struct-member</i> ;      Definition of one or more struct members for the struct subtype. ...

Use the **when** construct to create families of objects, in which multiple subtypes are derived from a common base struct type. A subtype is a struct type in which specific fields of the base struct have particular values, e.g.,

- a) If a struct type named `packet` has a field named `kind` that can have a value of `eth` or `atm`, then two subtypes of `packet` are `eth packet` and `atm packet`.
- b) If the `packet` struct has a Boolean field named `good`, two subtypes are `FALSE'good packet` and `TRUE'good packet`.
- c) Subtypes can also be combinations of fields, such as `eth TRUE'good packet` and `eth FALSE'good packet`.

Struct members defined in a **when** construct shall only be accessed in the subtype, not in the base struct. This provides a way to define a subtype that has some struct members in common with the base type and all of its other subtypes, but has other struct members that belong only to the current subtype.

If like inheritance is used to create a subtype of a base struct type, the base type shall not be extended by using **when**.

Syntax example:

```
struct packet {
    len : uint;
    good : bool;
    when FALSE'good packet {
        pkt_msg() is {
            out("bad packet")
        }
    }
}
```

## 6.7 Extending when subtypes

There are two general rules governing the extensions of **when** subtypes:

- a) If a struct member is declared in the base struct, it shall not be re-declared in any **when** subtype, but it can be extended.
- b) With the exception of coverage groups and the events associated with them, any struct member defined in a **when** subtype does not apply or is unknown in other subtypes, including:
  - 1) fields
  - 2) constraints
  - 3) events
  - 4) methods
  - 5) **on**
  - 6) **expect**
  - 7) **assume**

### 6.7.1 Coverage and when subtypes

All coverage events shall be defined in the base struct. Attempts to do so within a subtype, however, shall result in a load time error. Coverage groups shall be defined in the base struct or in the subtype.

### 6.7.2 Extending methods in when subtypes

A method defined or extended within a **when** construct is executed in the context of the subtype and can freely access the unique struct members of the subtype with no need for any casting.

When a method is declared in a base type, each extension of the method in a subtype shall have the same parameters and return type as the original declaration. Attempts to do otherwise shall result in a load time error. However, if a method is not declared in the base type, each definition of the method in a subtype can have different parameters and return type.

If more than one method of the same name is known in a **when** subtype, any reference to that method is ambiguous and shall result in a load-time error. To remove the ambiguity from such a reference, use the **as\_a()** type casting operator (see [5.8.1](#)) or the **when** subtype qualifier syntax.

Method calls are checked when the *e* code is parsed. If there is no ambiguity, the method to be called is selected and all similar references are resolved in the same manner.

See also [31.3.2](#).

## 6.8 Defining fields: field

<b>Purpose</b>	Define a struct field	
<b>Category</b>	Struct member	
<b>Syntax</b>	[ <b>package</b>   <b>protected</b>   <b>private</b> ] [ <b>const</b> ] [ <b>!</b> ] [ <b>%</b> ] <i>field-name</i> [: <i>type</i> ] [[ <i>min-val</i> .. <i>max-val</i> ]] [( <b>bits</b>   <b>bytes</b> ): <i>num</i> ]	
<b>Parameters</b>	<b>package</b>   <b>protected</b>   <b>private</b>	Sets any access restriction: which code, at what scope, can access this struct member. Otherwise, the default setting is: all code has access to this struct member. See <a href="#">23.3</a> for the keyword definitions.
	<b>const</b>	Denotes this field shall retain a constant value throughout its lifetime.
	<b>!</b>	Denotes an ungenerated field. The <b>!</b> and <b>%</b> options can be used together, in either order.
	<b>%</b>	Denotes a physical field. The <b>!</b> and <b>%</b> options can be used together, in either order.
	<i>field-name</i>	The name of the field being defined.
	<i>type</i>	The type for the field. This can be any scalar type, string, struct, or list. If the field name is the same as an existing type, the <b>: type</b> part of the field definition can be omitted. Otherwise, the type specification is required.
	<i>min-val..max-val</i>	An optional range of values for the field. If no range is specified, the range is the default range for the field's type.
	( <b>bits</b>   <b>bytes</b> : <i>num</i> )	The width of the field in bits or bytes. This syntax can be used for any scalar field, even if the field has a type with a known width.

This defines a field to hold data of a specific type. It can be a constant value (**const**), a physical field (**%**) or a virtual field (the default), and generated (the default) or not generated (**!**). For scalar data types, the size of the field can also be specified in bits or bytes.

Syntax example:

```
private const %packet : packet_t
```

### 6.8.1 Constant values

**const** identifies a field whose value is kept constant throughout the lifetime of the object and enforces this constant value. The *e* compiler may take advantage of **const** declarations to optimize memory use. A significant reduction in memory consumption may result from declaring a **when** determinant field (see [6.6](#)) as **const**.

#### 6.8.1.1 Initializing const fields

Initialization of fields declared as **const** shall be completed during the creation phase of a struct.

- a) A value for a **const** field can be generated during the generation of the enclosing struct.
- b) A value can be assigned to a **const** field:
  - 1) During generation, if the field is also marked as *do-not-generate*, using **pre\_generate()** (see [10.5.2](#)) or **post\_generate()** (see [10.5.3](#));

- 2) During unpacking, using **do\_unpack()** (see [20.4.1.1.2](#));
- 3) By using an assignment action while creating an object, with a **new...with** action block (see [4.16.2](#)) or the **init()** method (see [28.2.2.1](#)).

In these contexts, a **const** field of the newly created struct may be used on the left side of an assignment operator.

- c) A **const** field can also be initialized by using a built-in initialization mechanism, such as the **copy()** (see [28.4.1](#)) or **read\_binary\_struct()** method (see [30.5.2](#)).
- d) A **const** field can only be assigned a value once.

### 6.8.1.2 Restrictions

- Fields of **list** type (see [6.9](#)) cannot be declared **const**.
- Fields declared under **when** subtypes with a non-constant determinant cannot be declared **const**.
- Fields of **enum** types (see [5.7](#)) that are declared **const** have no default value upon creation of the struct (even if zero (0) is a possible enumerated value); they need to be initialized as specified in [6.8.1.1](#).
- Constant fields cannot be passed by reference.
- A **const** field can only be initialized with one of the constructs listed in [6.8.1.1](#).

### 6.8.2 Physical fields

A field defined as a physical field (with the % option) is packed when the struct is packed. Fields that represent data to be sent to the HDL device in the simulator or that are to be used for memories need to be physical fields. Nonphysical fields are called *virtual fields* and are not packed automatically when the struct is packed, although they can be packed individually.

If no range is specified, the width of the field is determined by the field's type. For a physical field, use the (**bits : num** or **bytes : num**) syntax to specify the width when the field's type does not have a known width.

### 6.8.3 Ungenerated fields

A field defined as ungenerated (with the ! option) is not generated automatically. This is useful for fields that are to be explicitly assigned during a test or whose values involve computations that cannot be expressed in constraints.

Ungenerated fields have default initial values (0 for scalars, NULL for structs, and an empty list for lists). An ungenerated field whose value is a range (such as [ 0 . . 100 ]) gets the first value in the range. If the field is a struct, its values remains NULL; therefore, the referenced struct is not allocated and none of the fields in it are generated.

### 6.8.4 Assigning values to fields

Unless a field is defined as ungenerated, a value is generated for it when the struct is generated, subject to any constraints that exist for the field. However, even for generated fields, values can be assigned in user-defined methods or predefined methods, such as **init()**, **pre\_generate()**, or **post\_generate()**. The ability to assign a value to a field is not affected by either the ! option or any generation constraints.

## 6.9 Defining list fields

This subclause defines list fields.

### 6.9.1 list of

<b>Purpose</b>	Define a list field	
<b>Category</b>	Struct member	
<b>Syntax</b>	<b>[!][%]</b> <i>list-name</i> <b>[<i>length-exp</i>]: list of <i>type</i></b>	
<b>Parameters</b>	<b>!</b>	Do not generate this list. The <b>!</b> and <b>%</b> options can be used together, in either order.
	<b>%</b>	Denotes a physical list. The <b>!</b> and <b>%</b> options can be used together, in either order.
	<i>list-name</i>	The name of the list being defined.
	<i>length-exp</i>	An expression that gives the initial size for the list. The expression shall evaluate to a non-negative integer.
	<i>type</i>	The type of items in the list. This can be any scalar type, string, or struct. It shall not be a list.

This defines a list of items of the specified type.

An initial size can be specified for the list; the list initially contains that number of items. The size shall conform to the initialization rules, the generation rules, and the packing rules. Even if an initial size is specified, the list size can change during a test if the list is operated on by a list method that changes the number of items.

All list items are initialized to their default values when the list is created. For a generated list, the initial default values are replaced by generated values. For information about initializing list items to particular values, see [5.3.3.6](#) and [10.2.7.3](#).

Syntax example:

```
packets : list of packet
```

## 6.9.2 list(key) of

<b>Purpose</b>	Define a keyed list field
<b>Category</b>	Struct member
<b>Syntax</b>	<b>![%]</b> <i>list-name</i> : <b>list(key: key-field) of</b> <i>type</i>
<b>Parameters</b>	<b>!</b> Do not generate this list. For a keyed list, the <b>!</b> is required, not optional.
	<b>%</b> Denotes a physical list. The <b>%</b> option can precede or follow the <b>!</b> .
	<i>list-name</i> The name of the list being defined.
	<i>key-field</i> The key of the list. For a list of structs, it is the name of a field of the struct. For a list of scalar or string items, it is the item itself, represented by the <b>it</b> variable. This is the field or value that the keyed list pseudo-methods check when they operate on the list.
	<i>type</i> The type of items in the list. This can be any scalar type, string, or struct. It shall not be a list.

Keyed lists are used to enable faster searching of lists by designating a particular field or value to use during the search. A keyed list can be used, for example, in the following ways:

- As a hash table, in which searching only for a key avoids the overhead of reading the entire contents of each item in the list.
- For a list that has the capacity to hold many items, but only contains a small percentage of its capacity, randomly spread across the range of possible items, e.g., a sparse memory implementation.

Besides the **key** parameter, the keyed list syntax differs from the regular list syntax in the following ways:

- a) The list shall be declared with the **!** do-not-generate operator. This means a keyed list needs to be built item-by-item, since it cannot be generated.
- b) The *[exp]* list size initialization syntax is not allowed for keyed lists, i.e., **list[exp]: list(key: key) of type** is not legal. Similarly, **keep** shall not be used to constrain the size of a keyed list.
- c) A keyed list is a distinct type, different from a regular list. This means a keyed list cannot be assigned to a regular list or vice versa, e.g., if `list_a` is a keyed list and `list_b` is a regular list, `list_a = list_b` shall result in an error.

If the same key value exists in more than one item in a keyed list, the keyed list pseudo-methods use the latest item in the list (the one with the highest list index number). Other items with the same key value are ignored. The keyed list pseudo-methods (see [27.7](#)) only work on lists that were defined and created as keyed lists. Conversely, restrictions apply when using regular list pseudo-methods or other operations on keyed lists (see [27.7.4](#)).

Syntax example:

```
!locations : list(key:address) of location
```

## 6.10 Projecting list of fields

<b>Purpose</b>	Specifying a field from all items in a list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list.field-name</i>	
<b>Parameters</b>	<i>list</i>	A list of structs.
	<i>field-name</i>	A name of a field or list in the struct type.

This returns a list containing the contents of the specified *field-name* for each item in the *list*. If the *list* is empty, it returns an empty list. This syntax is the same as *list.apply(field)* (see [27.4.1](#)).

An error shall be issued if the *list* is not a list of structs or the struct type does not have a field named *field-name*.

Syntax example:

```
s_list.fld_nm
```

## 6.11 Defining attribute fields

<b>Purpose</b>	Define the behavior of a field when copied or compared	
<b>Category</b>	Unit member	
<b>Syntax</b>	<b>attribute</b> <i>field-name attribute-name = exp</i>	
<b>Parameters</b>	<i>field-name</i>	The name of a field in the current struct.
	<i>attribute-name</i>	<i>attribute-name</i> is one of the following: <ul style="list-style-type: none"> <li>a) <b>deep_copy</b>—controls how the field is copied by the <b>deep_copy()</b> routine.</li> <li>b) <b>deep_compare</b>—controls how the field is compared by the <b>deep_compare()</b> routine.</li> <li>c) <b>deep_compare_physical</b>—controls how the field is compared by the <b>deep_compare_physical()</b> routine.</li> <li>d) <b>deep_all</b>—controls how the field is copied by the <b>deep_copy()</b> routine or compared by the <b>deep_compare()</b> or <b>deep_compare_physical()</b> routines.</li> </ul>
	<i>exp</i>	<i>exp</i> is one of the following: <ul style="list-style-type: none"> <li>a) <b>normal</b>—performs a deep (recursive) copy or comparison.</li> <li>b) <b>reference</b>—performs a shallow (non-recursive) copy or comparison.</li> <li>c) <b>ignore</b>—do not copy or compare.</li> </ul>

Defining attributes controls how a field behaves when it is copied or compared. These attributes are used by **deep\_copy()**, **deep\_compare()**, and **deep\_compare\_physical()**. For a full description of the behavior specified by each expression, see [29.1.1](#), [29.1.2](#), or [29.1.3](#), respectively.

To determine which attributes of a field are valid, all extensions to a unit or struct are scanned in the order they were loaded. If several values are specified for the same attribute of the same field, the last attribute specification loaded is the one that is used.

The **attribute** construct can appear anywhere, including inside a **when** construct or an **extend** construct.

Syntax example:

```
attribute channel deep_copy = reference
```



## 7. Units

This clause describes the constructs used to define units and their use in a modular verification methodology (see also [Clause 6](#)).

### 7.1 Overview

*Units* are the basic structural blocks for creating verification modules (*verification cores*) that can easily be integrated to test larger and larger portions of an HDL design as it develops. Units, like structs, are compound data types that contain data fields, procedural methods, and other members. Unlike structs, however, a unit instance is bound to a particular component in the DUT (an HDL path). Furthermore, each unit instance has a unique and constant place (an *e* path) in the runtime data structure of an *e* program. Both the *e* path and the complete HDL path associated with a unit instance are determined during pre-run generation.

The basic runtime data structure of an *e* program is a tree of unit instances whose root is **sys**, the only predefined unit in *e*. Additionally there are structs that are dynamically bound to unit instances. The runtime data structure of a typical *e* program is similar to that of the XYZ\_router program shown in [Figure 3](#).

Each unit instance in the unit instance tree of the XYZ\_router matches a module instance in the Verilog DUT, as shown in [Figure 4](#). The one-to-one correspondence in this particular design between *e* unit instances and DUT module instances is not required for all designs. In more complex designs, there can be several levels of DUT hierarchy corresponding to a single level of hierarchy in the tree of *e* unit instances.

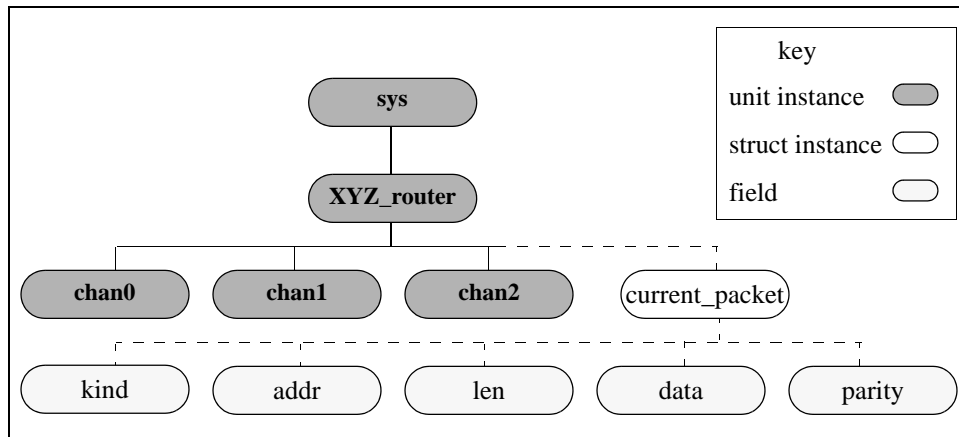


Figure 3—Runtime data structure of the XYZ\_Router

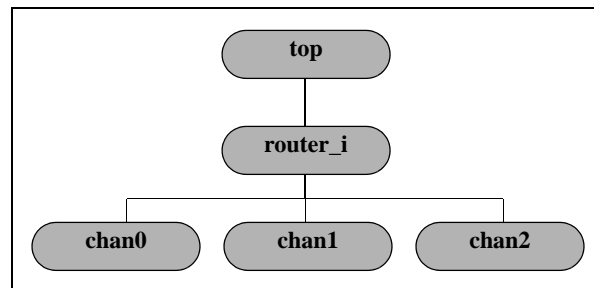


Figure 4—DUT router hierarchy

By binding an *e* unit instance to a particular component in the DUT hierarchy, signals can be referenced within that DUT component using relative HDL pathnames. When the units are integrated into a unit instance tree during pre-run generation, the complete pathname for each referenced HDL object is determined by concatenating the complete HDL path of the parent unit to the path of the unit containing the referenced object. This ability to use relative pathnames to reference HDL objects means the combination of verification cores can be changed as the HDL design and the verification environment evolve; regardless of where the DUT component is instantiated in the final integration, the HDL pathnames in the verification environment remain valid.

### 7.1.1 Units vs. structs

Modeling a DUT component with a unit or struct often depends on which verification strategy is employed. Compelling reasons for using a unit instead of a struct include the following:

- The DUT component will be tested both as a stand-alone and integrated into a larger system.  
Modeling the DUT component with a unit allows the use of relative pathnames when referencing HDL objects. When integrating the component with the rest of the design, simply change the HDL path associated with the unit instance and all the HDL references it contains are updated to reflect the component's new position in the design hierarchy. This methodology eliminates the need for computed HDL names (e.g., `'(path_str).sig'`), which can impact runtime performance.
- Methods that access many signals at runtime will be used.  
The correctness of all signal references within units is determined and checked during pre-run generation. If an *e* program does not contain user-defined units, the absolute HDL references within structs are also checked during pre-run generation. However, if it does contain user-defined units, the relative HDL references within structs are checked at runtime. In this case, using units rather than structs can enhance runtime performance.

On the other hand, using a struct to model abstract collections of data, such as packets, allows more flexibility in generating the data. With structs, the data can be generated during pre-run generation, at runtime, or on-the-fly (possibly in response to conditions in the DUT). Unit instances, however, can only be generated during pre-run generation, because each unit instance has a unique and constant place (an *e* path) in the runtime data structure of an *e* program, just as an HDL component instance has a constant place in the DUT hierarchical tree. Thus, the unit tree cannot be modified by generating unit instances on-the-fly.

NOTE—An allocated struct instance automatically establishes a reference to its parent unit. If this struct is generated during pre-run generation, it inherits the parent unit of its parent struct. If the struct is dynamically allocated by the **new** or **gen** action, then the parent unit is inherited from the struct belonging to the enclosing method.

### 7.1.2 HDL paths and units

Relative HDL paths are essential in creating a verification module that can be used to test a DUT component. Once an *e* unit instance is bound to a particular component in the DUT hierarchy, regardless of where the DUT component is instantiated in the final integration, the HDL pathnames are still valid.

#### Example

The XYZ\_router shown in [Figure 3](#) can be bound to the DUT router shown in [Figure 4](#) as follows.

- a) Use the **hdl\_path()** method within a **keep** constraint to associate a unit or unit instance with a DUT component. The following code extends **sys** by creating an instance of the XYZ\_router unit and binds the unit instance to the router\_i instance in the DUT.

```
extend sys {
  unit_core : XYZ_router is instance;
  keep unit_core.hdl_path() == "top.router_i"
```

- }  
 b) Similarly, the following code creates three instances of `XYZ_channel` in `XYZ_router` and constrains the HDL path of the instances to be `chan0`, `chan1`, and `chan2`. These are the names of the channel instances in the DUT relative to the `router_i` instance.

```
unit XYZ_router {
  channels : list of XYZ_channel is instance;
  keep channels.size() == 3;
  keep for each in channels {
    .hdl_path() == append("chan", index)
  }
}
```

The full path for a unit instance is used to resolve any internal HDL object references that contain relative HDL paths. It is determined during generation by appending the HDL path of the child unit instance to the full path of its parent, starting with `sys.sys` has the empty full path `" "`. The full path for the `XYZ_router` instance is `top.router_i` and that for the first channel instance is `top.router_i.chan0`.

NOTE—By default, the `do_print()` method of `any_unit` prints two predefined lines, as well as the user-defined fields. The predefined lines display the *e* path and the full HDL path for that unit. The *e* path line contains a hyperlink to the parent unit.

### 7.1.3 Methodology limitations and recommendations

Fields of type unit can be generated dynamically. However, the field shall be constrained to only refer to an existing unit instance.

The following limitations are implied by the nature of unit instances and fields of type unit:

- Unit instances cannot be the object of a **new** or **gen** action or a call to **copy()**.
- Unit instances cannot be placed on the LHS of the assignment operator.
- List methods that alter the original list, like **list.add()** or **list.pop()**, cannot be applied to lists of unit instances.
- Units are intended to be used as structural components and not as data carriers. Therefore, using physical fields in unit instances, as well as packing or unpacking into unit instances, is not recommended. Unpacking into a field of type unit when the field is NULL shall cause a runtime error.

To create a modular verification environment, the following recommendations are also important:

- a) For numeric settings, use **set\_config\_max()** for any global configuration options.
- b) Avoid global changes to the default packing options. Instead, define unit-specific options in the top-level unit and access them from lower-level units with **get\_enclosing\_unit()**.
- c) Place references to HDL objects in unit methods. To access HDL objects from struct methods, declare additional methods in a unit. When these access methods are one line of *e* code, declare them as **inline** methods for maximum efficiency.
- d) Use computed pathnames in structs that can be dynamically associated with more than one unit.
- e) Pre-run generation is performed before creating the `stubs` file (see [Clause 24](#)). To minimize the time required to create a `stubs` file, move any pre-run generation that is not related to building the tree of unit instances into the procedural code, preferably as an extension of the **run()** method of the appropriate structs.

## 7.2 Defining units and fields of type unit

The following subclauses describe the constructs for defining units and fields of type unit.

### 7.2.1 unit

<b>Purpose</b>	Define a data struct associated with an HDL component or block
<b>Category</b>	Statement
<b>Syntax</b>	<b>unit</b> <i>unit-type</i> [ <b>like</b> <i>base-unit-type</i> ] { [ <i>unit-member</i> ; ...]}
<b>Parameters</b>	<i>unit-type</i> The name of the unit.
	<i>base-unit-type</i> The name of the unit from which the new unit inherits its members.
	<i>unit-member</i> ; ...    The contents of the unit. Like structs, units can have the following types of members: <ul style="list-style-type: none"> <li>— data fields for storing data</li> <li>— methods for procedures</li> <li>— events for defining temporal triggers</li> <li>— coverage groups for defining coverage points</li> <li>— <b>when</b>, for specifying inheritance subtypes</li> <li>— declarative constraints for describing relations between data fields</li> <li>— <b>on</b>, for specifying actions to perform upon event occurrences</li> <li>— <b>expect</b>, for specifying temporal behavior rules</li> </ul> A unit can be empty, containing no members.

Because the base unit type (**any\_unit**) is derived from the base struct type (**any\_struct**), user-defined units have the same predefined methods. A unit type can be extended or used as the basis for creating unit subtypes. Extended unit types or unit subtypes inherit the base type's members and contain additional members (see also [7.1.1](#)).

Syntax example:

```

unit XYZ_channel {
    event external_clock;
    event packet_start is rise('valid_out')@sim;
    event data_passed;

    data_checker() @external_clock is {
        while 'valid_out' == 1 do {
            wait cycle;
            check that 'data_out' == 'data_in'
        };
        emit data_passed
    };

    on packet_start {
        start data_checker()
    }
}

```

### 7.2.2 field: unit-type is instance

<b>Purpose</b>	Define a unit instance field	
<b>Category</b>	Unit member	
<b>Syntax</b>	<i>field-name</i> [: <i>unit-type</i> ] <b>is instance</b>	
<b>Parameters</b>	<i>field-name</i>	The name of the unit instance being defined.
	<i>unit-type</i>	The name of a unit type. If the field name is the same as an existing type, the : <i>unit-type</i> part of the field definition can be omitted. Otherwise, the type specification is required.

This defines a field of a unit to be an instance of a unit type. Units can be instantiated within other units, thus creating a unit tree. The root of the unit tree is **sys**, the only predefined unit in *e*. The do-not-generate operator (!) is not allowed with fields of type unit instance. The physical field operator (%) is not allowed with fields of type unit instance. Instantiating a unit in a struct shall cause a compile-time error; units can only be instantiated within another unit.

Syntax example:

```
cpu : XYZ_cpu is instance
```

### 7.2.3 field: unit-type

<b>Purpose</b>	Define a field of type unit	
<b>Category</b>	Struct or unit member	
<b>Syntax</b>	[!] <i>field-name</i> [: <i>unit-type</i> ]	
<b>Parameters</b>	!	Denotes an ungenerated field. If this field is generated on-the-fly, it needs to be constrained to an existing unit instance or a runtime error shall occur.
	<i>field-name</i>	The name of the field being defined.
	<i>unit-type</i>	The name of a unit type. If the field name is the same as an existing type, the : <i>unit-type</i> part of the field definition can be omitted. Otherwise, the type specification is required.

This defines a field of unit type. A field of unit type is always either NULL or a reference to a unit instance of a specified unit type. If a field of type unit is generated, it shall be constrained to an existing unit instance.

Do not use the physical field operator (%) with fields of type unit.

Syntax example:

```
extend XYZ_router {
    !current_chan : XYZ_channel
}
```

### 7.2.4 field: list of unit instances

<b>Purpose</b>	Define a list field of unit instances	
<b>Category</b>	Struct or unit member	
<b>Syntax</b>	<i>name</i> <b>[</b> <i>length-exp</i> <b>]</b> : <b>list of unit-type is instance</b>	
<b>Parameters</b>	<i>name</i>	The name of the list being defined.
	<i>length-exp</i>	An expression that gives the initial size for the list.
	<i>unit-type</i>	A unit type.

This defines a list field of unit instances. A list of unit instances can only be created during pre-run generation and cannot be modified after it is generated. List operations, such as **list.add()** or **list.pop()**, that alter the list created during pre-run generation are not allowed for lists of unit instances.

Do not use the physical field operator (%) with lists of unit instance.

Syntax example:

```
channels : list of XYZ_channel is instance
```

### 7.2.5 field: list of unit-type

<b>Purpose</b>	Define a list field of type unit	
<b>Category</b>	Struct or unit member	
<b>Syntax</b>	<b>[!]</b> <i>name</i> <b>[</b> <i>length-exp</i> <b>]</b> : <b>list of unit-type</b>	
<b>Parameters</b>	<b>!</b>	Do not generate this list.
	<i>name</i>	The name of the list being defined.
	<i>length-exp</i>	An expression that gives the initial size for the list.
	<i>unit-type</i>	A unit type.

This defines a list field of type unit. An item of a list field of type unit is always either NULL or a reference to a unit instance of the specified unit type. If a list field of type unit is generated, either the entire list shall be constrained to an existing list of unit instances or each item in the list shall be constrained to an existing unit instance.

Do not use the physical field operator (%) with lists of unit type.

Syntax example:

```
var currently_valid_channels : list of XYZ_channel
```

## 7.3 Unit attributes

Units have attributes that affect their binding to a particular component in the DUT. Use the **attribute()** syntax to assign unit attributes in constraints, as follows:

**keep** [**soft**] [*unit-exp.*]**attribute()** == *value*

Use soft constraints for attributes that can be overridden.

Unit attributes can also be called as methods, in which case they shall return the value assigned to them in the pre-run generation phase.

### 7.3.1 hdl\_path()

<b>Purpose</b>	Specify a relative HDL path for a unit instance
<b>Category</b>	Unit attribute
<b>Syntax</b>	[ <i>unit-exp.</i> ] <b>hdl_path()</b> : string
<b>Parameters</b>	<i>unit-exp</i> An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed.

Specifies the HDL path of a unit instance. The most important role of this method is to bind a unit instance to a particular component in the DUT hierarchy. Binding an *e* unit or unit instance to a DUT component enables the referencing of signals within that component using relative HDL pathnames.

Regardless of where the DUT component is instantiated in the final integration, the HDL pathnames are still valid. The binding of unit instances to HDL components is a part of the pre-run generation process and needs to be done by using **keep** constraints. The HDL path for **sys** cannot be constrained.

NOTE—Although absolute HDL paths are allowed, relative HDL paths are recommended for using a modular verification strategy.

Syntax example:

```

extend dut_error_struct {
    write() is first {
        var channel : XYZ_channel =
            source_struct().try_enclosing_unit(XYZ_channel);

        if channel != NULL then {
            out("Error in XYZ channel: instance ", channel.hdl_path())
        }
    }
}

```

### 7.3.2 agent()

<b>Purpose</b>	Maps the DUT's HDL partitions into <i>e</i> code	
<b>Category</b>	Unit attribute	
<b>Syntax</b>	<i>[unit-exp.]agent():string</i>	
<b>Parameters</b>	<i>unit-exp</i>	An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed.

Specifying an agent by constraining the **agent()** attribute identifies the simulator and/or the HDL that is used to simulate the corresponding DUT component. Once a unit instance has an explicitly specified agent name then all other unit instances instantiated within it are implicitly bound to the same agent name, unless another agent is explicitly specified.

The following considerations also apply:

- The list of legal values to which the **agent()** attribute can be constrained is implementation-dependant, but the list needs to describe the various simulators and HDL languages supported by the implementation. It is suggested the values `verilog` and `vhdl` be part of this list.
- An agent name can be omitted in a single HDL environment, but it needs to be defined implicitly or explicitly in a mixed HDL environment for each unit instance that is associated with a non-empty **hdl\_path()**. If an agent name is not defined for a unit instance with a non-empty **hdl\_path()** in a mixed HDL environment, an error message shall be issued.
- Given the **hdl\_path()** and **agent()** constraints, a correspondence map is established between the unit instance HDL path and its agent name. Any HDL path below the path in the map is associated with the same agent, unless otherwise specified. This map is further used internally to pick the right adapter for each accessed HDL object.
- It is possible to access Verilog signals from a VHDL unit instance code and vice versa. Every signal is mapped to its HDL domain according to its full path, regardless of the specified agent of the unit from which the signal is accessed.
- When the **agent()** method is called procedurally, it returns the agent of the unit. The spelling of the agent string is exactly as specified in the corresponding constraint. An unsupported agent name shall cause an error message during the test phase.

NOTE—Agents are bound to unit instances during the generation phase. Consequently, there is no way to map between HDL objects and agents before generation. As a result of this limitation, HDL objects in a mixed Verilog/VHDL environment cannot be accessed before generation from **sys.setup()**.

Syntax example:

```
router : XYZ_router is instance;
  keep router.agent() == "verilog"
```

## 7.4 Predefined methods of any\_unit

There is a predefined generic type **any\_unit**, which is derived from **any\_struct**. **any\_unit** is the base type implicitly used in user-defined unit types, so all predefined methods for **any\_unit** are available for any user-defined unit. The predefined methods for **any\_struct** are also available for any user-defined unit.



### 7.4.1 full\_hdl\_path()

<b>Purpose</b>	Return an absolute HDL path for a unit instance	
<b>Category</b>	Predefined method of <b>any_unit</b>	
<b>Syntax</b>	<i>[unit-exp.]full_hdl_path()</i> : string	
<b>Parameters</b>	<i>unit-exp</i>	An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed.

Returns the absolute HDL path for the specified unit instance. This method is used mainly in informational messages.

Syntax example:

```
out("Mutex violation in ", get_unit().full_hdl_path())
```

### 7.4.2 e\_path()

<b>Purpose</b>	Returns the location of a unit instance in the unit tree	
<b>Category</b>	Predefined method of <b>any_unit</b>	
<b>Syntax</b>	<i>[unit-exp.]e_path()</i> : string	
<b>Parameters</b>	<i>unit-exp</i>	An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed.

Returns the location of a unit instance in the unit tree. This method is used mainly in informational messages.

Syntax example:

```
out("Started checking ", get_unit().e_path())
```

### 7.4.3 get\_parent\_unit()

<b>Purpose</b>	Return a reference to the unit containing the current unit instance	
<b>Category</b>	Predefined method of <b>any_unit</b>	
<b>Syntax</b>	<i>[unit-exp.]get_parent_unit()</i> : any_unit	
<b>Parameters</b>	<i>unit-exp</i>	An expression that returns a unit instance. If no expression is specified, the current unit instance is assumed.

Returns a reference to the unit containing the current unit instance.

Syntax example:

```
out(sys.unit_core.channels[0].get_parent_unit())
```

## 7.5 Unit-related predefined methods of **any\_struct**

This subclause describes the predefined methods of **any\_struct**.

### 7.5.1 **get\_unit()**

<b>Purpose</b>	Return a reference to the unit
<b>Category</b>	Predefined method of <b>any_struct</b>
<b>Syntax</b>	<i>[exp.]</i> <b>get_unit()</b> : any_unit
<b>Parameters</b>	<i>exp</i> An expression that returns a struct or unit. If no expression is specified, the current struct or unit is assumed.

When applied to an allocated struct instance, this method returns a reference to the parent unit—the unit to which the struct is bound. When applied to a unit, it returns the unit itself.

Any allocated struct instance automatically establishes a reference to its parent unit. If this struct is generated during pre-run generation, it inherits the parent unit of its parent struct. If the struct is dynamically allocated by the **new** or **gen** action, then the parent unit is inherited from the struct to which the enclosing method belongs.

This method is useful to determine the parent unit instance of a struct or unit. It can also be used to access predefined unit members, such as **hdl\_path()** or **full\_hdl\_path()**. To access user-defined unit members, use [7.5.2](#).

Syntax example:

```
out("Mutex violation in ", get_unit().full_hdl_path())
```

### 7.5.2 **get\_enclosing\_unit()**

<b>Purpose</b>	Return a reference to nearest unit of specified type
<b>Category</b>	Predefined pseudo-method of <b>any_struct</b>
<b>Syntax</b>	<i>[exp.]</i> <b>get_enclosing_unit(unit-type)</b> : unit-type
<b>Parameters</b>	<i>exp</i> An expression that returns a struct or unit. If no expression is specified, the current struct or unit is assumed. <sup>a</sup>
	<i>unit-type</i> The name of a unit type or unit subtype.

<sup>a</sup>If **get\_enclosing\_unit()** is called from within a unit of the same type as *exp*, it returns the present unit instance and not the parent unit instance.

Returns a reference to the nearest higher-level unit instance of the specified type, so fields of the parent unit can be accessed in a typed manner. The parent unit can be used to store shared data or options such as packing options that are valid for all its associated subunits or structs.

The unit type is recognized according to the same rules used for the **is a** operator (see [4.16.1](#)), i.e., if a base unit type is specified and an instance of a unit subtype exists, the unit subtype is found. If a unit instance of the specified type is not found, a runtime error shall be issued.

Syntax example:

```
unpack(p.get_enclosing_unit(XYZ_router).pack_config,
      'data', current_packet)
```

### 7.5.3 try\_enclosing\_unit()

<b>Purpose</b>	Return a reference to nearest unit of specified type or NULL
<b>Category</b>	Predefined pseudo-method of <b>any_struct</b>
<b>Syntax</b>	<i>[exp.]try_enclosing_unit(unit-type): unit-type</i>
<b>Parameters</b>	<i>exp</i> An expression that returns a struct or unit. If no expression is specified, the current struct or unit is assumed. <sup>a</sup>
	<i>unit-type</i> The name of a unit type or unit subtype.

<sup>a</sup>If **try\_enclosing\_unit()** is called from within a unit of the same type as *exp*, it returns the present unit instance and not the parent unit instance.

Like **get\_enclosing\_unit()** (see [7.5.2](#)), this method returns a reference to the nearest higher-level unit instance of the specified type, so fields of the parent unit can be accessed in a typed manner.

Unlike **get\_enclosing\_unit()**, this method does not issue a runtime error if no unit instance of the specified type is found. Instead, it returns NULL. This feature makes the method suitable for use in extensions to global methods, such as **dut\_error\_struct.write()**, which can be used with more than one unit type.

Syntax example:

```
var MIPS := source_struct().try_enclosing_unit(MIPS)
```

### 7.5.4 set\_unit()

<b>Purpose</b>	Change the parent unit of a struct
<b>Category</b>	Predefined method of <b>any_struct</b>
<b>Syntax</b>	<i>[struct-exp.]set_unit(parent: exp)</i>
<b>Parameters</b>	<i>struct-exp</i> An expression that returns a struct. If no expression is specified, the current struct is assumed.
	<i>parent</i> An expression that returns a unit instance.

Changes the parent unit of a struct to the specified unit instance. This method shall emit an error for units because the unit tree cannot be modified.

Syntax example:

```
p.set_unit(sys.unit_core)
```

## 7.6 Unit-related predefined routines

The predefined routines that are useful for units include `set_config_max()` and `get_all_units()`.

### 7.6.1 `set_config_max()`

<b>Purpose</b>	Increase values of numeric global configuration parameters	
<b>Category</b>	Predefined routine	
<b>Syntax</b>	<b>set_config_max</b> ( <i>category</i> : keyword, <i>option</i> : keyword, <i>value</i> : exp [, <i>option</i> : keyword, <i>value</i> : exp...])	
<b>Parameters</b>	<i>category</i>	Is one of the following: <b>cover</b> , <b>gen</b> , <b>memory</b> , and <b>run</b> , or any additional implementation-dependent categories.
	<i>option</i>	<p>The valid <b>cover</b> option is: <b>absolute_max_buckets</b>.</p> <p>The valid <b>generate</b> options are:</p> <ul style="list-style-type: none"> <li>— <b>absolute_max_list_size</b></li> <li>— <b>max_depth</b></li> <li>— <b>max_structs</b></li> </ul> <p>The valid <b>memory</b> options are:</p> <ul style="list-style-type: none"> <li>— <b>gc_threshold</b></li> <li>— <b>gc_increment</b></li> <li>— <b>max_size</b></li> <li>— <b>absolute_max_size</b></li> </ul> <p>The valid <b>run</b> option is: <b>tick_max</b>.</p> <p>The implementation can also introduce additional options.</p>
	<i>value</i>	The valid values for each option are implementation specific.

This routine sets the numeric options of a particular category to the specified maximum values.

NOTE—When working in a modular verification environment, it is recommended to use `set_config_max()` instead of `set_config()` (see [29.9](#)) in order to avoid possible conflicts that can happen in an integrated environment.

Syntax example:

```
set_config_max(memory, gc_threshold, 100M)
```

### 7.6.2 `get_all_units()`

<b>Purpose</b>	Return a list of instances of a specified unit type	
<b>Category</b>	Pseudo-routine	
<b>Syntax</b>	<b>get_all_units</b> ( <i>unit-type</i> ): list of <i>unit-type</i>	
<b>Parameters</b>	<i>unit-type</i>	The name of a unit type. The type needs to be defined or an error shall occur.

This routine receives a unit type as a parameter and returns a list of instances of this unit type, as well as any unit instances of any subtype of this unit type.

Syntax example:

```
print get_all_units(XYZ_channel)
```



## 8. Template types

This clause describes the principles and usage of *e* template features. Template types in *e* let you define generic structs and units that are parameterized by type, similar to C++ templates. You can later instantiate them, giving specific types as actual parameters.

NOTE— Because units are a special case of structs, you can define templates for both structs and units. In general, a template instance is a struct type. Provided it is legal, a template instance becomes a struct as soon as it is referenced. It can be used in any context, and in any way in which a regular struct can be used.

### 8.1 Defining a template type

<b>Purpose</b>	Define a template type
<b>Category</b>	Action
<b>Syntax</b>	<code>[<b>package</b>] <b>template</b> ( <b>struct</b>   <b>unit</b> ) <i>template-name</i> <b>of</b> ( <i>param-list</i> ) [<b>like</b> <i>base-type</i>] { [<i>member</i>;... ] }</code>
<b>Parameters</b>	<b>package</b> Denotes <b>package</b> access restriction to this template.
	<b>struct</b> or <b>unit</b> Denotes whether the instances of this template are structs or units.
	<i>template-name</i> The name of the template.
	<i>param-list</i> The template parameters (see <a href="#">8.1.1</a> ). Currently, only type parameters (<type>) are supported.
	<i>base-type</i> The base struct from which instances of the template inherit. The base type can itself be parameterized over one or more of the template parameters, in which case each template instance inherits from a different type (see <a href="#">8.1.2</a> ).
	<i>member</i> ;... The body of the template, which contains fields, methods, events, and other struct members. The template parameters can be used within the members as appropriate (see <a href="#">8.1.3</a> ).

NOTE— Template names share the namespace with types, so these names cannot be the same as any other type or template name in the same package. Templates are treated the same as types with respect to:

- Name resolution
- Access control

#### Template Definition Example

The following example defines a template struct that maps keys to values. The template has two type parameters. The first parameter, <key'type>, is the map key type, and the second parameter, <value'type>, is the value type. When this template is instantiated, the occurrences of the two parameters inside the template body are replaced by the actual types for that instance.

```
template struct map of (<key'type>, <value'type>) {
  keys: list of <key'type>;
  values: list of <value'type>;
  put(k: <key'type>, v: <value'type>) is {
    ...
  };
}
```

```

    get(k: <key'type>): <value'type> is {
        ...
    }
}

```

See [8.2](#) for an example of how this template would be used.

### 8.1.1 About template type parameters

A template definition contains a comma-separated list of *template parameters*. A parameter name must have the form `<[tag']type>`.

A type parameter can be any legal type in *e*. When the template is instantiated, a specific type is substituted for each such parameter. Inside the template body, a type parameter can occur at any place where a type is allowed or expected. In [Template Definition Example on page 107](#), both `<key'type>` and `<value'type>` are used to specify the types of fields, method parameters, and method return values.

### 8.1.2 Specifying a template base type

A simple way to create a template base type is to specify a concrete type as the base type, so that all of the template instances inherit from the same type. The base type can be a regular struct, or an instance of another template:

```

struct s { ... };
template struct t1 of <type> like s { ... };
template struct t2 of <type> like t1 of int { ... }

```

Another example derives a template from another template:

```

template ordered_set of <type> like set of <type> {...}

```

As for units in general, template units cannot derive from non-unit types, either regular or template.

As for regular structs and units, the default base type for struct templates is **any\_struct**, and the default base type for unit templates is **any\_unit**.

### 8.1.3 Template body

The template body consists of struct members. It can contain fields, methods, events, coverage groups, constraints, and any other kind of struct members. You can use a template parameter wherever a type is allowed.

#### *Template body example*

```

template struct packet of (<kind'type>, <data'type>) {
    size: uint;
    data1: <data'type>;
    data2: <data'type>;
    kind: <kind'type>;
    keep size < 256;
    sum_data(): <data'type> is {
        return data1 + data2
    }
}

```



### 8.1.4 Template types and when subtypes

You can define a **when** subtype within a template in the same way that you define a **when** subtype within a regular struct. For example, the following can be added to the packet template in [Template body example on page 108](#), assuming that <kind'type> has the value “red”:

*Example — when subtype*

```
template struct packet of (<kind'type>, <data'type>) {
    ...
    when red packet of (<kind'type>, <data'type>) {
        red_data: <data'type>
    }
}
```

### 8.2 Instantiating a template type

<b>Purpose</b>	Instantiate a template
<b>Category</b>	Action
<b>Syntax</b>	<i>template-name</i> <b>of</b> ( <i>actual-param-list</i> )
<b>Parameters</b>	<i>template-name</i> The name of the template.
	<i>actual-param-list</i> A comma-separated list of actual parameters for the template. You must give a legal type name for each type parameter. If a template has only one parameter, you can omit the parentheses around the single actual parameter.

A template instance creates a new struct, by substituting an actual parameter for the corresponding template parameter within the template body. This substitution also occurs in the definition of the base type.

NOTE— All instances of a template are considered to be defined where the template is defined, regardless of where they are instantiated. This applies, in particular, to name resolution and access control issues.

In general, you can use any legal type as an actual type parameter, including another template instance, or another instance of the same template. For example, given the map template in [Template Definition Example on page 107](#), you can create the following instances:

```
map of (int, int)
map of (s1, s2)
map of (s, map of (string, int))
```

Given the packet template in [Template body example on page 108](#), you can create the following instances:

```
packet of (color, int)
packet of (color, uint(bits: 64))
```

Not every template instance is legal. For example, the following two instances are illegal, because the code in the template body implies that <kind'type> must be an enum that has the value “red”:

```
-- Illegal template instances
packet of (int, int)
packet of ([green, blue], int)
```

Any two instances of a template, if their actual parameters refer to exactly the same types, are considered to be the same instance, even if syntactically they are different. For example, in the following code, fields `x` and `y` have the same type:

```
#define N 32;
type color: [red, green];
struct s {
    x: packet of (color, int(bits: 64));
    y: packet of (color, int(bits: N*2))
}
```

Because a template instance becomes a struct, its name can be used anywhere in the code where a struct name is allowed or expected.

### 8.2.1 Template subtype instances

If a template definition includes **when** subtypes, you can refer to it in the same way you refer to regular **when** subtypes. For example, given the packet template in [Example — when subtype on page 109](#), the following is a legal type name, because it denotes the red'kind subtype of the “packet of (color, int)” template instance.

```
red packet of (color, int)
```

### 8.2.2 Forward references

The rules for forward referencing of template definitions are the same as for type definitions. A template definition can refer to types, fields, and so on that are not yet defined at that point, but are defined later in the same translation unit<sup>11</sup>.

Similarly, a struct or template definition can refer to instances of templates that are defined later in the same translation unit. In particular, a template definition, like a struct definition, can refer to itself in its own body.

For example, the following code is legal:

```
template struct t1 of <type> {
    x: t2 of list of <type>;
};
template struct t2 of <type> {
    y: t1 of s;
};
struct s {
    f(): list of t1 of int is { ... }
}
```

## 8.3 Template types and reflection

Template instances, when created, are in fact types, so you can use them in reflection like any other type. With regard to the type name, the canonical name of the instance is used. For example:

```
var rf_s: rf_struct = rf_manager.get_struct_by_name("packet of (int,int)");
print rf_s.get_name();
rf_s = rf_manager.get_struct_by_name("packet of int(bits:8*2),int");
print rf_s.get_name();
```

prints:

<sup>11</sup>A single module, or a group of modules that are translated together due to cyclic import.

```
packet of (int, int)
packet of (int(bits: 16), int)
```

NOTE— If a template instance has not been used, it does not exist in the type system, so it is not represented in the reflection mechanism.

## 8.4 Template errors

Certain kinds of compilation errors are discovered and reported when processing the template definition. Others are discovered and reported upon specific instantiations.

### 8.4.1 Template definition errors

Two kinds of compilation errors are discovered and reported when a template definition is processed:

- Syntax errors
- Semantic errors related to type names within the template definition

If these kinds of errors are found within a template definition, they are reported in the same way as when they are in a regular struct definition. In particular, each error message refers to the source line where the error is present.

### 8.4.2 Template instantiation errors

Most kinds of semantic errors are discovered when the template is instantiated. Some instances of the same template can be legal, while others can be illegal and lead to semantic errors, except for those semantic errors related to type names used in the code ([8.4.1](#)).

An error message reporting a template instantiation error lists the source lines of the error itself and the source line in which the problematic template instantiation is first created. This report can be recursive—for example, if the template is instantiated within another template, its own instantiation line is also referred to, and so on.<sup>12</sup>

For example, trying to load the following code:

```
<'
template struct err_struct of (<first'type>, <second'type>) {
  x: <first'type>(bits: 32); // => <first'type> must be a scalar type
  y: list of <second'type>; // => <second'type> must not be a list type
};
extend sys {
  run() is also {
    var x: err_struct of (string, list of int);
  };
}
'>
```

will cause the following two error messages:

```
*** Error: 'string' is not a scalar type
        at line 3 in err.e
x: <first'type>(bits: 32);
```

<sup>12</sup> If any of the code references—the actual erroneous code or one of the template instantiations—originates in a macro, the macro source line is also listed, as for any other macro-related error messages.

```

        from template instantiation 'err_struct of (string,list of int)'
        at line 8 in err.e
    var x: err_struct of (string, list of int);

*** Error: Cannot define a list of lists
        at line 4 in err.e
y: list of <second'type>;
        from template instantiation 'err_struct of (string,list of int)'
        at line 8 in err.e
    var x: err_struct of (string, list of int);

```

If several different template instantiations cause errors on the same line of the template definition, all are reported, rather than reporting only one error per line.

### 8.4.3 Run-time errors

A run-time error related to a template definition is reported like any other run-time error. Only the actual error line for interpreted code, or the method containing it for compiled code, is reported, without reference to the template instantiations. This format applies to all kinds of run-time errors, including user errors, DUT errors, assertion failures, and so on.

## 8.5 Limitations

The following is a list of known limitations related to templates:

- A template instance cannot be created interactively from a command. If a specific instance of a template has not been referenced in the compiled/loaded code, it cannot be used in a command. Trying to do so causes an error message.
- A template instance cannot be created by the reflection mechanism at run-time. If a specific instance of a template has not been referenced in the compiled/loaded code, the reflection mechanism considers it to be a nonexistent type.
- A template method may not be declared as C routine.
- Once a template is defined, it can later—in the same module, or in a later loaded/compiled module—have its direct or indirect base type extended. For some instances, however, the members of the base struct extension might be applied earlier than the members of the original template definition, although they appear later in the code, so the order of the member is incorrect.

Therefore, the behavior for template instances is undefined, and might be different for different instances of the same template. A `WARN_EXTEND_AFTER_TEMPLATE` message is issued when a template instance is created that has the incorrect order for some of its methods, events, or expects. If you get this notification, make sure there is no real problem with this order.

NOTE—This behavior might change in the future, so do not rely on it if the order matters.

## 8.6 Templates vs. macros

In some sense, templates are similar to define-as macros. Both define parameterized code, which is instantiated by substituting parameters. In principle, a `<statement>` macro can be used for that purpose, if it has `<type>` syntactic arguments and a struct definition in its replacement code. Calling that macro will create a specific struct, which is analogous to creating a template instance. See [16](#) for information on macros.

However, there are several important differences that make using templates preferable:

- A macro must be instantiated explicitly in the code. Multiple explicit instantiations of the same macro causes an error, because the instances are multiple definitions of the same struct. Templates are instantiated automatically, the first time a specific instance is referred to in the code.
- A template definition is fully syntactically analyzed, and any syntax errors within a template definition are reported when the template is defined ([8.4.1](#)). A macro definition, in contrast, is not syntactically analyzed, and most syntax errors—except basic errors such as unbalanced parentheses in the code—are discovered only when the macro is called.
- With regard to name resolution and related issues, template code is treated in the context of the template definition, regardless of where a given template instance is referred to for the first time. Macro code, in contrast, is treated in the context of the macro call itself. For instance, if the macro replacement code tries to access the package-accessible field of a struct defined in the same package, the macro call will not compile if it resides in a different package.



## 9. *e* ports

This clause describes ports, *e* unit members that enhance the portability and interoperability of verification environments by making separation between an *e* unit and its interface possible.

### 9.1 Introduction to *e* ports

A *port* is an *e* unit member that makes a connection between an *e* unit and its interface to another internal or external entity. There are two ways to use ports:

- Internal ports (*e2e* ports) connect an *e* unit to another *e* unit.
- External ports connect an *e* unit to a simulated object.

External ports are a generic way to access simulated objects of various kinds. An external port is bound to a simulated object, e.g., an HDL signal in the DUT. Then all access to that signal is made via the port. The port can be used to access a different signal simply by changing the binding; all the code that reads or writes to the port remains the same. Similarly, port semantics remain the same, regardless of what simulator is used. A *simulator* is any hardware or software agent that runs in parallel with an *e* program and models the behavior of any part of the DUT or its environment.

#### 9.1.1 Creating port instances

A *port type* is defined by the following aspects:

- a) The kind of port: simple port, buffer port, event port, or method port.
  - 1) *Simple ports* access data directly.
  - 2) *Buffer ports* implement an abstraction of queues, with blocking **get()** and **put()**.
  - 3) *Event ports* transfer events between *e* units or between an *e* unit and a simulator.
  - 4) *Method ports* enable a regular or TCM defined in an *e* unit or a foreign programming language module to be called from another *e* unit or foreign programming language module.
- b) Direction, either input or output (or inout for simple and event ports).
- c) Data element, the *e* type that can be passed through this port.

Ports can only be instantiated within units using a unique instance name and the port type (direction, port kind, and a kind-specific type specifier). Like units, port instances are generated during pre-run generation and cannot be created, modified, or removed during a run.

The generic syntax for ports is:

*port-instance-name* : [*direction*] *port-kind* [**of** *type-specifier*] **is instance**

Event ports do not have a type specifier.

#### *Examples*

The following unit member creates a port instance:

```
data_in : in buffer_port of packet is instance
```

where

- The port instance name is *data\_in*
- The port kind is a buffer port

- The port direction is input
- The data element the port accepts is packet

As another example, the following line creates a list of simple ports that each pass data of type bit:

```
ports : list of simple_port of bit is instance
```

### 9.1.2 Using ports

A port's behavior is influenced by port attributes, such as **hdl\_path()** or **bind()**, that are applied to port instances using pre-run generation **keep** constraints. For example, the following lines of code create a port named data and connect (bind) it to an external simulator-related object whose HDL pathname is data.

```
data : inout simple_port of list of bit is instance;
keep bind(data, external);
keep data.hdl_path() == "data"
```

Each port kind has predefined methods that can be used to access the port values. For example, buffer ports have a predefined method **put()**, which writes a value onto an output port, as follows:

```
data_out : out buffer_port of cell is instance;

drive_all() @sys.any is {
    var stimuli : cell;
    var counter : int = 0;

    while counter < cells do {
        wait [1]*cycle;
        gen stimuli;
        data_out.put(stimuli);
        counter += 1
    }
}
```

### 9.1.3 Using port values and attributes in constraints

Like units, port instances can be created only during pre-run generation. They cannot be created by using **new** or generated at runtime. Consequently, a port value cannot be initialized or sampled in pre-run generation constraints. Port values can be used in on-the-fly generation constraints, in accordance with the basic constraint principles, such as the bidirectional nature of constraints.

## 9.2 Using simple ports

*Simple ports* can be used to transfer one data element at a time to or from an external simulated object, such as a Verilog register, a VHDL signal, a SystemC field, or an internal object (another *e* unit). A simple port's direction can be either input, output, or inout.

Use the **\$** port access operator to read or write port values. To access MVL on simple ports, either declare a port's data element to be mvl or list of mvl, or use the MVL methods. See [9.2.1](#) and [9.2.2](#) for more information.

Internal and external ports shall have a **bind()** attribute that defines how they are connected. In addition, the **delayed()** attribute can be used to control whether new values are propagated immediately or at the next tick.



An external simple port needs to have an `hdl_path()` attribute to specify the name of the object to which it is connected. In addition, an external simple port can have several additional attributes that enable continuous driving of external signals (see [9.7](#)).

### 9.2.1 Accessing simple ports and their values

Ports are containers, and the values they hold are separate entities from the port itself. The `$` access operator distinguishes port value expressions from port reference expressions.

The `$` operator, e.g., `p$`, can also be used to access or update the value held in a simple port `p`. When used on the RHS, `p$` refers to the port's value. On the LHS of an assignment, `p$` refers to the value's location, so an assignment to `p$` changes the value held in the port.

Without the `$` operator, an expression of any type port refers to the port itself, not to its value. In particular, an expression without the `$` operator can be used for operations involving port references.

#### Examples

##### Accessing port values

<code>print p\$</code>	Prints the value of a simple port, <code>p</code> . <sup>a</sup>
<code>p\$ = 0</code>	Assigns the value 0 to a simple port, <code>p</code> . <sup>b</sup>
<code>force p\$= 0</code>	Forces a simple external port to 0.
<code>print q\$[1:0]</code>	Prints the two lists of the value of <code>q</code> .

<sup>a</sup>Compare with `print p`, which prints information about port `p`.

<sup>b</sup>Compare `p$ = 0;` with `pref = NULL`, which modifies a port reference so it does not point to any port instance.

##### Accessing a port

<code>print p</code>	Prints the information about port <code>p</code> . Port <code>p</code> is defined as: <code>p: simple_port of int (bits:8) is instance</code>
<code>keep q == p</code>	<code>q</code> refers to the port instance <code>p</code> . Port reference <code>q</code> is defined as: <code>!q: simple_port of int (bits:8)</code>

### 9.2.2 MVL on simple ports

There are two ways to read and write MVL on simple ports, as follows:

- Define a port and use the predefined MVL methods described in [9.9](#) to read and write values to the port.
- Define ports of type `mvl` or list of `mvl` and use the `$` access operator to read and write the port values.

Ports of type `mvl` or list of `mvl` (MVL ports) allow easy transformation between exact *e* values and MVL, which is useful for communicating with objects that sometimes model bit values other than 0 or 1 during a test. Otherwise, using non-MVL ports is preferable, since they allow keeping the port values in a bit-by-bit representation, while MVL ports require having an *e* list for an MVL vector. MVL type definition and MVL functions are described in [9.9](#).

The Verilog comparison operators (`===` or `!==`) cannot be used with numeric ports or MVL ports. These operators can be used only with the tick access syntax.

### 9.2.3 @sim temporal expressions with external simple ports

Specifying an event port causes *e* to be sensitive to the corresponding HDL signal during the entire simulation session. This might result in some unnecessary runtime performance cost if *e* only needs to be sensitive in certain scenarios. In such cases, use an external simple port in TEs with **@sim** instead. The syntax is:

**[change | rise | fall] (simple-port\$)@sim**

Typically, this syntax is used in wait actions.

#### Example

```
transaction_complete : in simple_port of bit is instance;
  keep bind(transaction_complete, external);

write_transaction(data: list of byte) @clk$ is {
  //...
  data_port$ = data;
  wait rise(transaction_complete$)@sim
}
```

Trying to apply the **@sim** operator to a bound internal port shall cause an error when the corresponding TE is evaluated, which occurs at runtime.

## 9.3 Using buffer ports

*Buffer ports* can be used to insert data elements into a queue or extract elements from a queue. Data is inserted and extracted from the queue in first-in-first-out (FIFO) order. When the queue is full, write-access to the port is blocked. When the queue is empty, read-access to the port is blocked. The queue size is fixed during generation by the **buffer\_size()** attribute and cannot be changed at runtime. The queue size can be set to 0 for rendezvous ports. See [9.7.2.2](#) and [9.3.1](#) for more information.

A buffer port's direction can be either input or output. Use the buffer port's predefined **get()** and **put()** methods to read or write port values. These methods are *time-consuming methods* (TCMs). The **\$** port access operator cannot be used with buffer ports.

Buffer ports shall have a **bind()** attribute that defines how they are connected. In addition, the **delayed()** attribute can be used to control whether new values are propagated immediately or at the next tick. The **pass\_by\_pointer()** attribute controls how data elements of composite type are passed. See also [9.7](#).

### 9.3.1 Rendezvous-zero size buffer queue

In rendezvous-style handshaking protocol, access to a port is blocked after each **put()** until a subsequent **get()** is performed, and access is blocked after each **get()** until a subsequent **put()** is performed.

This style of communication is easily achieved by using buffer ports with a data queue size of 0. The following example shows how this is done.

#### 9.3.2 Example

```
unit consumer {
  in_p : in buffer_port of atm_cell is instance
}
unit producer {
```

```

    out_p : out buffer_port of atm_cell is instance
  };
  extend sys {
    consumer : consumer is instance;
    producer : producer is instance;
    keep bind(producer.out_p, consumer.in_p);
    keep producer.out_p.buffer_size() == 0
  }

```

## 9.4 Using event ports

*Event ports* can be used to transfer events between two *e* units or between an *e* unit and an external object. An internal event port's direction can be either input, output, or inout. Use the **\$** port access operator to read or write port values (see [9.4.1](#)).

Internal and external ports need to have a **bind()** attribute that defines how they are connected. An external port needs to have an **hdl\_path()** attribute to specify the name of the object to which it is connected. The **edge()** attribute for an external input event port specifies the edge on which an event is generated. See also [9.7](#).

### 9.4.1 Accessing event ports

Use the **\$** access operator to access the event associated with an event port. An expression of type **event\_port** without the **\$** operator refers to the port itself and not to its event.

### 9.4.2 Example

This example shows how to connect event ports [using a **bind()** constraint] and use the **\$** operator to access event ports in event contexts.

```

unit u1 {
  in_ep : in event_port is instance;
  tcml1()@in_ep$ is {
    // ...
  }
};

unit u2 {
  out_ep : out event_port is instance;
  counter : uint;
  event clk is @sys.any;

  on clk {
    counter = counter + 1;
    if counter %10 == 0 then {
      emit out_ep$
    }
  }
};

extend sys {
  u1 : u1 is instance;
  u2 : u2 is instance;
  keep bind(u1.in_ep, u2.out_ep)
}

```

## 9.5 Using method ports

*Method ports* can be used to either call or export methods and TCMs defined in other *e* units or in foreign programming language modules. The advantages of method ports are:

- A transaction-level interface can be implemented between *e* and a high-level model described in a foreign language.
- The decision about which method to call (e.g., an *e* method or a foreign function) can be postponed from compile time to pre-run generation.

### 9.5.1 Method types

A method port shall be parameterized by a type of a special kind—a method type. The *method type* specifies the prototype (signature) of the method and implies specific user-defined semantics. For example, the following declares a method type for a method that accepts two integer arguments and returns an integer:

```
method_type adder_method_t(arg1:int, arg2:int): int
```

The following method type declaration has the same prototype as `adder_method_t`, but implies different user-defined semantics:

```
method_type local_adder_method_t(arg1:int, arg2:int): int
```

A method type that is associated with a TCM shall be defined with the **@sys.any** sampling event, e.g.,

```
method_type send_packet_method_t(p:packet)@sys.any
```

Method types shall be defined with a unique name; this name shall be explicitly specified in the instance declaration of the method port (see [9.6.4](#)). For example, the following associates the `add` method port with the `adder_method_t` method type:

```
add : out method_port of adder_method_t is instance
```

The method type has semantic implications for a port beyond the simple matching of parameters and result types; it is also used to clarify runtime messages related to a particular method port. Thus, two method ports cannot be bound just because they have the same signature; they also need to be associated with the same method type.

### 9.5.2 Input method ports

An *input method port* declares an *e* method as callable from another *e* unit or from a foreign agent. The method port instance shall:

- a) Reside in the same unit as its associated method;
- b) Have an instance name that matches the name of the associated method;
- c) Have a method type that matches the prototype of the associated method.

The method type and its prototype match if:

- 1) They have the same number of parameters.
  - 2) Any parameters are of the same types (and in the right order).
  - 3) Any return values are of the same type.
- d) Include the **@sys.any** sampling event (in the method type declaration) if the method type is associated with a TCM.

### 9.5.3 Output method ports

*Output method ports* can be used to call regular or TCMs defined in other *e* units or written in foreign programming languages.

### 9.5.4 Invoking method ports

The `$()` access operator can be used to call the method port (see also [9.6.9](#)). The rules for parameter type checking, TCM call requirements, etc., are the same as those for invoking an *e* method directly. In particular, TCM method ports can only be called from inside a TCM scope.

The parameter passing semantics are the same as in direct calls to *e* methods. Scalar parameters are passed by value, while composite parameters (struct or list types) are passed by reference.

Other considerations:

- Do not rely on the ability to modify separate fields or list elements of the incoming parameter in the actual method. Instead, use the return value (or explicit passing of parameters by reference).
- All ports are elaborated after the end of **post\_generate()**. Thus, method ports cannot be invoked before generation or from constraints.
- Calling an empty-bound method port is equivalent to calling an empty *e* method.

### 9.5.5 Binding method ports

If a set of input and output ports are bound, all the ports are connected (no matter how the binding pairs were specified) and a change on any output port affects all input ports. While this makes sense for simple ports, which are used to emulate wires, it does not for method ports. For example, if there are two output method ports, `Ao` and `Bo`, three input method ports, `Ai`, `Bi`, and `ABi`, and the binding looks like:

```
bind(Ao, Ai);
bind(Bo, Bi);
bind(Ao, ABi);
bind(Bo, ABi)
```

the intention probably is that a call to `Ao` causes a call of `Ai` and `ABi`, while a call to `Bo` causes a call of `Bi` and `ABi`. This intention is implemented; however, a call to `Ao` also causes a call of `Bi`, and a call to `Bo` also causes a call of `Ai`.

To bind multiple output ports to a common input, define the common input as a list of in method ports (see [9.6.4](#)). Then, each of the input method ports is associated with the method via the list name.

*Example*

The list of in method ports is

```
type src_t : [A, B];
method_type p_t(s:src_t);

extend sys {
  Ao : out method_port of p_t is instance;
  Bo : out method_port of p_t is instance;

  ABi : list of in method_port of p_t is instance;
  keep ABi.size() == 2;

  ABi(src: src_t) is {
    out("AB(", src, ")")
  }
}
```

and the binding is:

```
// each output also invokes the common input
keep bind(Ao, ABi[0]);
keep bind(Bo, ABi[1]);

run() is also {
    Ao$(A);
    Bo$(B)
}
```

## 9.6 Defining and referencing ports

This subclause details how to define or reference a port.

### 9.6.1 simple\_port

<b>Purpose</b>	Access other port instances or external simulated objects directly	
<b>Category</b>	Unit member	
<b>Syntax</b>	<i>port-instance-name</i> : [list of] [ <i>direction</i> ] <b>simple_port of element-type is instance</b>	
<b>Parameters</b>	<i>port-instance-name</i>	A unique identifier used to reference the port or access its value.
	<i>direction</i>	One of <b>in</b> , <b>out</b> , or <b>inout</b> . The default is <b>inout</b> , which means values can be read from and written to this port. For an <b>in</b> port, values can only be read from the port; for an <b>out</b> port, values can only be written to the port.
	<i>element-type</i>	Any predefined or user-defined <i>e</i> type, except a port type or unit type.

Simple ports can be used to transfer one data element at a time to or from an external simulated object or internal object (another *e* unit). External ports can transfer scalar types and lists of scalar types, including MVL data elements. Structs or lists of structs cannot be passed through external simple ports.

The port can be configured to access a different signal simply by changing the binding; all the code that reads or writes to the port remains the same. Similarly, port semantics remain the same, regardless of what simulator is used. Binding is fixed during generation.

A simple port's direction can be either in, out, or inout. Omitting the direction is the same as writing inout. Port types with different direction are not equivalent. The following types are fully equivalent:

```
data :      simple_port of byte is instance;
data : inout simple_port of byte is instance
```

*Syntax example:*

```
data : in simple_port of byte is instance
```

### 9.6.2 buffer\_port

<b>Purpose</b>	Implement an abstraction of queues with blocking get and put	
<b>Category</b>	Unit member	
<b>Syntax</b>	<i>port-instance-name</i> : [ <b>list of</b> ] <i>direction</i> <b>buffer_port of element-type is instance</b>	
<b>Parameters</b>	<i>port-instance-name</i>	A unique identifier used to reference the port or access its value.
	<i>direction</i>	One of <b>in</b> or <b>out</b> . There is no default. For an <b>in</b> port, values can only be read from the port; for an <b>out</b> port, values can only be written to the port. See <a href="#">9.8</a> for information on how to read and write buffer ports.
	<i>element-type</i>	Any predefined or user-defined <i>e</i> type, except a port type or a unit type.

Buffer ports can be used to insert data elements into a queue or extract elements from a queue. Data is inserted and extracted from the queue in FIFO order. When the queue is full, write-access to the port is blocked. When the queue is empty, read-access to the port is blocked.

The queue size is fixed during generation by the **buffer\_size()** attribute and cannot be changed at runtime. The queue size can be set to 0 for rendezvous ports.

Use the buffer port's predefined **get()** and **put()** methods to read or write port values. These methods are TCMs. The \$ port access operator cannot be used with buffer ports.

A typical usage of a buffer port is in a *producer* and *consumer* protocol, where one object puts data on an output port at possibly irregular intervals and another object with the corresponding input port reads the data at its own rate.

*Syntax example:*

```
rq : in buffer_port of bool is instance
```

### 9.6.3 event\_port

<b>Purpose</b>	Transfer events between units or between simulators and units	
<b>Category</b>	Unit member	
<b>Syntax</b>	<i>event-port-field-name</i> : [ <b>list of</b> ] [ <i>direction</i> ] <b>event_port is instance</b>	
<b>Parameters</b>	<i>event-port-field-name</i>	A unique identifier used to reference the port or access its value.
	<i>direction</i>	One of <b>in</b> , <b>out</b> , or <b>inout</b> . The default is <b>inout</b> , which means events can be emitted and sampled on the port. For a port with direction <b>in</b> , events can only be sampled. For a port with direction <b>out</b> , events can only be emitted.

Event ports can be used to transfer events between two *e* units or between an *e* unit and an external object. Use the \$ port access operator to read or write port values (see [9.4.1](#)).

An event port's direction can be either in, out, or inout. Omitting the direction is the same as writing inout. Port types with different directions are not equivalent. The following types are fully equivalent:

```
clk :          event_port is instance;
clk : inout event_port is instance
```

In addition, the following are not allowed:

- Using the **on** struct member for event ports
- Coverage on event ports
- Specifying a temporal formula (e.g., out event\_port is ...) to define an out event port

It is possible, however, to define an additional event and connect it to the event port, e.g.,

```
ep : in event_port is instance;
    keep bind(ep, external);
event e is @ep$
```

*Syntax example:*

```
clk : in event_port is instance
```

#### 9.6.4 method\_port

<b>Purpose</b>	Enable invocation of abstract functions	
<b>Category</b>	Unit member	
<b>Syntax</b>	<i>port-instance-name</i> : [ <b>list of</b> ] <i>direction</i> <b>method_port of</b> <i>method-type</i> <b>is instance</b>	
<b>Parameters</b>	<i>port-instance-name</i>	A unique identifier used to reference the method port or invoke the actual method. For input method ports, this name shall be the same as that of the associated method.
	<i>direction</i>	One of <b>in</b> or <b>out</b> . There is no default. For an <b>in</b> port, only the method to activate can be specified; for an <b>out</b> port, the method can be invoked.
	<i>method-type</i>	A <i>method type</i> that specifies the port semantics (see also <a href="#">9.6.5</a> ).

Method ports implement an abstraction of the calling methods (time-consuming or not) in other units or external agents, while delaying the binding from compile time to pre-run generation time.

*Syntax example:*

```
convert_string : out method_port of str2uint_method_t is instance
```



### 9.6.5 `method_type` *method-type-name*

<b>Purpose</b>	Associate method prototype with type name and enable notification	
<b>Category</b>	Statement	
<b>Syntax</b>	<b>method_type</b> <i>method-type-name</i> ([ <i>param-list</i> ]) [: <i>return-type</i> ] [ <b>@sys.any</b> ]	
<b>Parameters</b>	<i>method-type-name</i>	A unique identifier used to reference the method type.
	<i>param-list</i>	This needs to match the parameter list of the <i>e</i> method or external function.
	<i>return-type</i>	This needs to match the return type of the <i>e</i> method or external function.
	<b>@sys.any</b>	If the method type declaration includes the <b>@sys.any</b> sampling event, this method type can only be used for method ports associated with a TCM.

A method port (see 9.6.4) shall be parameterized by a *method type*, which specifies the prototype (signature) of the method. The method type name can also be included in any runtime messages related to a specific method port.

*Syntax example:*

```
method_type str2uint_method_t(s:string): uint
```

### 9.6.6 Port reference

<b>Purpose</b>	Reference a port instance	
<b>Category</b>	Unit field, variable, or method parameter	
<b>Syntax</b>	<b>[!   var]</b> <i>port-reference-name</i> [: <i>direction</i> ] <i>port-kind</i> [ <b>of</b> <i>element-type</i> ]	
<b>Parameters</b>	<i>port-reference-name</i>	A unique identifier.
	<i>direction</i>	One of <b>in</b> or <b>out</b> ; for simple ports and event ports, this can also be <b>inout</b> .
	<i>port-kind</i>	One of <b>simple_port</b> , <b>buffer_port</b> , or <b>event_port</b> .
	<i>element-type</i>	Required if port-kind is <b>simple_port</b> or <b>buffer_port</b> .

If a port reference is a field, then it shall be marked as non-generated or it needs to be constrained to an existing port instance. Otherwise, a generation error shall result.

*Syntax example:*

```
!in_int_buffer_port_ref : in buffer_port of int
```

### 9.6.7 Port: \$

<b>Purpose</b>	Read or write a value to a simple port or event port	
<b>Category</b>	Operator	
<b>Syntax</b>	<i>exp</i> <b>\$</b>	
<b>Parameters</b>	<i>exp</i>	An expression that returns a simple port or event port instance.

The **\$** access operator can be used to access or update the value held in a simple port or event port. When used on the RHS, *p***\$** refers to the port's value. On the LHS of an assignment, *p***\$** refers to the value's location, so an assignment to *p***\$** changes the value held in the port.

Without the **\$** operator, an expression of any type port refers to the port itself, not to its value. In particular, an expression without the **\$** operator can be used for operations involving port references.

*Syntax example:*

```
p$ = 32'bz           // Assigns an mvl literal to the port 'p'
```

### 9.6.8 Method port reference

<b>Purpose</b>	Reference a method port instance	
<b>Category</b>	Unit field, variable, or method parameter	
<b>Syntax</b>	<b>[!   var]</b> <i>port-reference-name</i> <b>:</b> <i>direction</i> <b>method_port of method-type</b>	
<b>Parameters</b>	<i>port-reference-name</i>	A unique identifier used to reference the method port.
	<i>direction</i>	One of <b>in</b> or <b>out</b> .
	<i>method-type</i>	A <i>method type</i> that specifies the port semantics (see also <a href="#">9.6.5</a> ).

Method port instances may be referenced by a field, variable, or method parameter of the same port type.

If a port reference is a field, it shall be marked as non-generated or it needs to be constrained to an existing port instance. Otherwise, a generation error shall result. Also, port binding is allowed only for port instance fields, not for port reference fields (see also [9.5.5](#)).

*Syntax example:*

```
!in_method_port_ref : in method_port of burst_method_t
```

### 9.6.9 Method port: \$

<b>Purpose</b>	Call an out method port	
<b>Category</b>	Operator	
<b>Syntax</b>	<i>port-exp</i> <b>\$</b> ( <i>out-method-port-param-list</i> )	
<b>Parameters</b>	<i>port-exp</i>	An expression that returns an output method port instance.
	<i>out-method-port-param-list</i>	A list of actual parameters to the output method port. The number and type of the parameters, if any, shall match the <i>method type</i> (see also <a href="#">9.6.5</a> ).

The \$ access operator can be used to call an output method port. An attempt to call a method via the port without using the \$ operator shall result in a syntax error. Without the \$ operator, an expression of any type port refers to the port itself, not to its value. In particular, an expression without the \$ operator can be used for operations involving port references.

*Syntax example:*

```
u = convert_string$("32")    //calls the convert_string out method port
```

## 9.7 Port attributes

Ports have attributes that affect their behavior and how they can be used. Use the *attribute()* syntax to assign port attributes in pre-generation constraints, as follows:

```
keep [soft] port_instance.attribute() == value
```

Use soft constraints for attributes that can be overridden.

Most port attributes are ignored, unless the port is an external port, but it does no harm to specify attributes for ports that are not external ports. Attributes intended for external ports do not have to be supported for a particular simulator.

### 9.7.1 Generic port attributes

Port attributes that are potentially valid for all simulators are described in [Table 21](#). However, a particular simulator adapter might not implement some of these attributes. Depending on the simulator adapter, port attributes might cause additional code to be written to the *stubs* file (see [Clause 24](#)). In that case, if an attribute is added or changed, the *stubs* file needs to be rewritten.

### 9.7.2 Port attributes for HDL simulators

Port attributes that are potentially valid for all HDL simulators are described in [Table 22](#). However, a particular simulator adapter might not implement some of these attributes. The port attributes in [Table 22](#) enable extended functionality. They cause additional information to be written into the HDL *stubs* file to enhance user control over the driving of HDL signals. For this reason, any attribute shown in [Table 22](#) is added or changed, the *stubs* file needs to be rewritten.

*Example*

The following attributes define a port that is eight bits wide; read operations occur with one-unit delay; drive operations have a five-unit delay:

Table 21—Generic port attributes

Attribute	Description	Applies to
<b>bind()</b>	Connects two internal ports or connect a port to an external object. Type: <i>bool</i> Default: none See also <a href="#">9.7.2.1</a> .	All kinds of internal and external ports
<b>buffer_size()</b>	Specifies the maximum number of elements for a buffer port queue. Type: <b>uint</b> Default: none See also <a href="#">9.7.2.2</a> .	Buffer ports
<b>declared_range()</b>	Specifies the bit width of an external multi-bit object. Type: <i>string</i> Default: none See also <a href="#">9.7.2.3</a> .	External output simple ports that are bound to some kinds of multi-bit objects
<b>delayed()</b>	Specifies whether propagation of a new port value assignment occurs immediately or is delayed to the tick boundary. Type: <i>bool</i> Default: TRUE See also <a href="#">9.7.2.4</a> .	Internal and external simple ports
<b>driver()</b>	When TRUE, an additional resolved HDL driver is created for the corresponding simulator item, and that driver is written to instead of the port. Type: <i>bool</i> Default: FALSE See also <a href="#">9.7.2.5</a> .	External output simple ports
<b>driver_delay()</b>	Specifies the delay time for all assignments from <i>e</i> to the port. Type: <b>time</b> Default: 0 See also <a href="#">9.7.2.6</a> .	External output simple ports
<b>edge()</b>	Specifies the edge on which an event is generated. Type: <b>event_port_edge</b> Default: <b>change</b> See also <a href="#">9.7.2.8</a> .	External input event ports
<b>hdl_convertor()</b>	Specifies the rules for converting method port arguments between <i>e</i> and a foreign language, such as SystemVerilog or VHDL. The syntax of the string value associated with <b>hdl_convertor()</b> is defined by the language adapter itself. Type: <i>string</i> Default: none	Method ports
<b>hdl_path()</b>	Specifies a relative path of the corresponding simulated item as a string. Type: <i>string</i> Default: none See also <a href="#">9.7.2.9</a> .	External ports

**Table 21—Generic port attributes (continued)**

Attribute	Description	Applies to
<b>pack_options()</b>	Specifies how the port's data element is implicitly packed and unpacked. Type: <b>pack_options</b> Default: NULL See also <a href="#">9.7.2.10</a> .	External simple ports
<b>pass_by_pointer</b>	When TRUE, composite data (structs or lists) are passed by reference. Type: <i>bool</i> Default: FALSE (pass by value) See also <a href="#">9.7.2.11</a> .	Internal simple or buffer ports whose data element is a composite type (lists and structs)

```

data : inout simple_port of uint(bits:8) is instance;
  keep bind(data, external);
  keep data.hdl_path()      == "sig";
  keep data.declared_range() == "[7:0]";
  keep data.verilog_strobe() == "#1";
  keep data.verilog_drive()  == "#5"

```

**Table 22—Port attributes for Verilog or VHDL agents**

Attribute	Description	Applies to
<b>driver_initial_value()</b>	Applies an initial mvl value to the port. Type: <i>list of mvl</i> Default: {} (empty list) See also <a href="#">9.7.2.7</a> .	External output simple ports
<b>verilog_drive()</b>	Specifies the event on which the data is driven to the Verilog object. Type: <i>string</i> Default: none See also <a href="#">9.7.2.12</a> .	External output simple ports
<b>verilog_drive_hold()</b>	Specifies an event after which the port data is set to Z. Type: <i>string</i> Default: none See also <a href="#">9.7.2.13</a> .	External output simple ports
<b>verilog_forcible()</b>	Allows forcing of Verilog wires. Type: <i>bool</i> Default: FALSE See also <a href="#">9.7.2.14</a> .	External output simple ports
<b>verilog_strobe()</b>	Specifies the sampling event for the Verilog signal that is bound to the port. Type: <i>string</i> Default: none See also <a href="#">9.7.2.15</a> .	External output simple ports
<b>verilog_wire()</b>	Binds an external out port to a Verilog wire. Type: <i>bool</i> Default: FALSE See also <a href="#">9.7.2.16</a> .	External output simple ports

Table 22—Port attributes for Verilog or VHDL agents (*continued*)

Attribute	Description	Applies to
<b>vhdl_delay_mode()</b>	Specifies whether pulses whose period is shorter than the delay are propagated through the driver. Type: <b>vhdl_delay_mode</b> Default: <b>TRANSPORT</b> (all pulses, regardless of length, are propagated) See also <a href="#">9.7.2.17</a> .	External output simple ports
<b>vhdl_driver()</b>	This is an alias for the <b>driver()</b> attribute. Type: <i>bool</i> Default: <b>FALSE</b> See also <a href="#">9.7.2.5</a> .	External output simple ports

### 9.7.2.1 bind()

<b>Purpose</b>	Connect two internal ports or connect a port to an external object	
<b>Category</b>	Generic port attribute	
<b>Syntax</b>	<b>bind</b> ( <i>exp1</i> , <i>exp2</i> ) <b>bind</b> ( <i>exp1</i> , ( <b>external</b>   <b>empty</b>   <b>undefined</b> ))	
<b>Parameters</b>	<i>exp1</i> , <i>exp2</i>	One or more expressions of port type. If two expressions are given and the port types are compatible, the two port instances are connected.
	<b>external</b>	Defines a port as connected to a simulated object, such as a Verilog register, VHDL signal, or SystemC object.
	<b>empty</b>	Defines a disconnected port. Runtime accessing of a port with an empty binding is allowed.
	<b>undefined</b>	Defines a disconnected port. Runtime accessing of a port with an undefined binding shall cause an error.

Ports are connected to other *e* ports or to external simulated objects, such as Verilog registers, VHDL signals, or SystemC methods, using a pre-run generation constraint on the **bind()** attribute. Ports can also be left explicitly disconnected by using **empty** or **undefined**.

#### 9.7.2.1.1 Rules

- a) All ports shall be bound in one of the following ways:
  - 1) A set of bound ports must include at least two ports, one of which is an input or inout port, and the other an output or inout port.
  - 2) Bound to an external simulated item.
  - 3) Explicitly disconnected (empty or undefined).
- b) Only ports of the same kind can be bound together. A simple port cannot be bound to a buffer port or an event port, and a buffer port cannot be bound to an event port.
- c) Dangling ports [ports without **bind()** attributes] shall cause an error during elaboration (see [9.7.2.1.2](#)).

- d) A port can be explicitly disconnected and then overridden with a binding to an internal or external object.
- e) All ports connected together shall have the exact same element type.

### 9.7.2.1.2 Checking of ports

Binding and checking of ports takes place automatically at the end of the predefined **generate\_test()** test method. This process, called *elaboration of ports*, includes checking for dangling ports and binding consistency (directions, buffer sizes, and so on).

A port that has no **bind()** constraint is a *dangling port*. Since all ports need to be bound, a dangling port shall cause an elaboration-time error.

### 9.7.2.1.3 Disconnected ports

A port that is bound using the **empty** or **undefined** keyword is called a *disconnected port*. The **empty** or **undefined** keyword can only appear as the second argument of the **bind()** constraint, in place of a second port instance name.

Empty binding can be used to define a port that is connected to nothing. Runtime accessing of an empty-bound port is allowed. Its effect depends on the operation and type of the port.

- Reading from an empty-bound simple port returns the last written value or the default of the port element type, if no value has been written so far.
- Writing to an empty-bound output or inout simple port stores the new value internally.
- Reading from an empty-bound buffer port causes the thread to halt.
- Writing to an empty-bound buffer port causes the thread to halt if the buffer is full.
- Waiting for an empty-bound event port causes the thread to halt. If the port direction is inout, then emitting the port resumes the thread.
- An empty-bound event port can be emitted.

A subsequent constraint can be used to overwrite the empty binding constraint.

Like empty binding, undefined binding can define a port that is connected to nothing. The difference is runtime accessing of a port with an undefined binding shall cause an error.

A subsequent constraint can be used to overwrite the undefined binding constraint.

*Syntax example:*

```
buf_in1 : in buffer_port of int(bits:16) is instance;
  keep bind(buf_in1, empty)
```

### 9.7.2.2 buffer\_size()

<b>Purpose</b>	Specify the size of a buffer port queue	
<b>Category</b>	Buffer port attribute	
<b>Syntax</b>	<i>exp.buffer_size() == num</i>	
<b>Parameters</b>	<i>exp</i>	An expression of type <b>[in   out] buffer_port of type</b> .
	<i>num</i>	An integer specifying the maximum number of elements for the queue.

This attribute determines the number of **put()** actions that can be performed before a **get()**. A **get()** action is required to remove data and make more room in the queue. Specifying a buffer size of 0 means rendezvous-style synchronization.

No default buffer size is provided. If a buffer size is not specified in a constraint, an error shall occur. It is only necessary to specify a buffer size for one of the two ports in a pair of connected ports. That size applies to both ports. If the two ports have different buffer sizes specified, then both of them get the larger of the two sizes.

Syntax example:

```
keep u.p.buffer_size() == 20
```

### 9.7.2.3 declared\_range()

<b>Purpose</b>	Specify the bit width of a multi-bit external object	
<b>Category</b>	External port attribute	
<b>Syntax</b>	<i>exp.declared_range() == string</i>	
<b>Parameters</b>	<i>exp</i>	An expression of a simple port type.
	<i>string</i>	A string that is a valid range expression, e.g., " [msb:lsb] "

This string attribute is meaningful for external simple ports that are bound to multi-bit objects. Because it is legal to bind a port to an HDL object with a different size, the range information is not extracted from the port declaration. In order to implement access to multi-bit signals correctly in the `stubs` file (see [Clause 24](#)), this attribute is required when using the **verilog\_wire()**, **verilog\_drive()**, **verilog\_strobe()**, or **driver()** attributes.

The interpretation of the string is simulator-specific.

Syntax example:

```
keep u.p.declared_range() == "[31:0]"
```



### 9.7.2.4 delayed()

<b>Purpose</b>	Specify immediate or delayed propagation of new values	
<b>Category</b>	Simple port attribute	
<b>Syntax</b>	<code>exp.delayed()</code> <code>not exp.delayed()</code> <code>exp.delayed() == bool</code>	
<b>Parameters</b>	<i>exp</i>	An expression of a simple port type.
	<i>bool</i>	Either TRUE or FALSE. The default is TRUE.

This Boolean attribute specifies whether propagation of a new port value assignment occurs immediately or is delayed. When the **delayed()** attribute is TRUE (the default), propagation of external ports is delayed until the next tick. Propagation of internal ports is delayed until the next tick when the **sys.time** value changes. This behavior is consistent with the definition of delayed assignments in *e* and matches temporal *e* semantics with regard to the multiple ticks occurring at the same simulator time.

To make assigned values on ports visible immediately, constrain this attribute to be FALSE.

Syntax example:

```
keep not u.p.delayed()
```

### 9.7.2.5 driver()

<b>Purpose</b>	Create a resolved driver for an external object	
<b>Category</b>	External out port attribute	
<b>Syntax</b>	<code>exp.driver()</code> <code>not exp.driver()</code> <code>exp.driver() == bool</code>	
<b>Parameters</b>	<i>exp</i>	An expression of a simple port type.
	<i>bool</i>	Either TRUE or FALSE. The default is FALSE.

This Boolean attribute is meaningful only for external out ports. When this attribute is set to TRUE, an additional resolved HDL driver is created for the corresponding simulator item and that driver is written to instead of the port.

Every port instance associated with the same simulator can create a separate driver, thus allowing HDL resolution to be applied for multiple *e* resources.

Syntax example:

```
keep u.p.driver()
```

### 9.7.2.6 driver\_delay()

<b>Purpose</b>	Specify the delay for assignments to a port	
<b>Category</b>	External out simple port attribute	
<b>Syntax</b>	<i>exp.driver_delay() == time</i>	
<b>Parameters</b>	<i>exp</i>	An expression of a simple port type.
	<i>time</i>	A value of type <b>time</b> (64 bits). The default is 0.

This attribute is meaningful only for external out ports. It specifies the delay time for all assignments from *e* to the port. This attribute is silently ignored, unless the **driver()** attribute or the **vhdl\_driver()** attribute is set to TRUE.

*Syntax example:*

```
keep u.p.driver_delay() == 2
```

### 9.7.2.7 driver\_initial\_value()

<b>Purpose</b>	Specify an initial value for an HDL object	
<b>Category</b>	HDL port attribute	
<b>Syntax</b>	<i>exp.driver_initial_value() == mvl-list</i>	
<b>Parameters</b>	<i>exp</i>	An expression that returns a port instance.
	<i>mvl-list</i>	A lists of mvl values. The default is {} (an empty list).

This *mvl-list* type attribute applies an initial mvl value to an external Verilog or VHDL object. This attribute is silently ignored, unless the **driver()** attribute or the **vhdl\_driver()** attribute is set to TRUE.

The default value of this attribute is MVL\_X.

*Syntax example:*

```
keep u.p.driver_initial_value() == {MVL_X; MVL_X; MVL_1; MVL_1}
```

### 9.7.2.8 edge()

<b>Purpose</b>	Specify the edge on which an event is generated	
<b>Category</b>	Event port attribute	
<b>Syntax</b>	<i>exp</i> . <b>edge()</b> == <i>edge-option</i>	
<b>Parameters</b>	<i>exp</i>	An expression of a <b>buffer_port</b> type.
	<i>edge-option</i>	A value of type <b>event_port_edge</b> .

This attribute of type **event\_port\_edge** (for an external event port) specifies the edge on which an event is generated. The possible values are:

- a) **change, rise, fall**—equivalent to the behavior of @**sim** TEs. This means that transitions between *x* and 0, *z*, and 1 are not detected; *x* to 1 is considered a rise; *z* to 0 a fall, and so on.
- b) **any\_change**—any change within the supported MVL values is detected, including transitions from *x* to 0 and 1 to *z*.
- c) **MVL\_0\_to\_1**—transitions from 0 to 1 only.
- d) **MVL\_1\_to\_0**—transitions from 1 to 0 only.
- e) **MVL\_X\_to\_0**—transitions from *x* to 0 only.
- f) **MVL\_0\_to\_X**—transitions from 0 to *x* only.
- g) **MVL\_Z\_to\_1**—transitions from *z* to 1 only.
- h) **MVL\_1\_to\_Z**—transitions from 1 to *z* only.

The default is **change**.

*Syntax example:*

```
keep e.edge() == any_change
```

### 9.7.2.9 hdl\_path()

<b>Purpose</b>	Map port instance to an external object	
<b>Category</b>	Generic port attribute	
<b>Syntax</b>	<i>exp</i> . <b>hdl_path()</b> == <i>string</i>	
<b>Parameters</b>	<i>exp</i>	An expression of a port type.
	<i>string</i>	A string specifying the path to the external object. The default is an empty string.

This attribute specifies a path for accessing an external, simulated object. The path is a concatenation of the partial paths for the port itself and its enclosing units. The partial paths can use any supported separator. To allow portability between simulators, use the *e* canonical path notation.

*Syntax example:*

```
clk : in event_port is instance;
keep clk.hdl_path() == "clk"
```

### 9.7.2.10 pack\_options()

<b>Purpose</b>	Specify how an external port's data element is implicitly packed and unpacked	
<b>Category</b>	External simple port attribute	
<b>Syntax</b>	<i>exp.pack_options() == pack-option</i>	
<b>Parameters</b>	<i>exp</i>	An expression of a simple or buffer port type.
	<i>pack-option</i>	A predefined or user-defined pack option. The default is NULL.

This attribute can be used to specify the way that data element of external ports is implicitly packed and unpacked. This attribute exists both for units and ports, and can be propagated downwards from an enclosing unit instance to its ports and other unit instances.

*Syntax example:*

```
keep u.p.pack_options() == packing.low_big_endian
```

### 9.7.2.11 pass\_by\_pointer()

<b>Purpose</b>	Specify how composite data is transferred by internal ports	
<b>Category</b>	Internal port attribute	
<b>Syntax</b>	<i>exp.pass_by_pointer()</i> <b>not</b> <i>exp.pass_by_pointer()</i> <i>exp.pass_by_pointer() == bool</i>	
<b>Parameters</b>	<i>exp</i>	An expression of a simple or buffer port type.

This Boolean attribute specifies how composite data (structs or lists) is transferred by internal simple ports or buffer ports. By default, this attribute is `FALSE` and complex objects are deep-copied upon an internal port access operation. To pass data by reference and speed up the test, set this attribute to `TRUE` (and verify no test-correctness violations exist).

*Syntax example:*

```
keep u.p.pass_by_pointer();
keep not u.p.pass_by_pointer();
```

### 9.7.2.12 verilog\_drive()

<b>Purpose</b>	Specify timing control for data driven to a Verilog object
<b>Category</b>	Verilog port attribute
<b>Syntax</b>	<i>exp.verilog_drive() == timing-control</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>timing-control</i> A string specifying any legal Verilog timing control (event or delay).

This string attribute tells an external output port to drive its data to a Verilog signal when the specified timing occurs. This can be a Verilog TE, such as `@(posedge top.clk)`, or a simple unit delay, e.g., `#1`.

*Syntax example:*

```
keep u.p.verilog_drive() == "@posedge clk2"
```

### 9.7.2.13 verilog\_drive\_hold()

<b>Purpose</b>	Specify when to set the port to Z
<b>Category</b>	Verilog port attribute
<b>Syntax</b>	<i>exp.verilog_drive_hold() == string</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>string</i> A string specifying any legal Verilog timing control.

On the first occurrence of the specified event after the port data is driven, the value of the corresponding Verilog signal is set to z. The event is a string specifying any legal Verilog timing control. The **verilog\_drive()** attribute (see [9.7.2.12](#)) needs to be specified before using this attribute.

*Syntax example:*

```
keep u.p.verilog_drive_hold() == "@negedge clk2"
```

#### 9.7.2.14 verilog\_forcible()

<b>Purpose</b>	Specify a Verilog object can be forced
<b>Category</b>	Verilog port attribute
<b>Syntax</b>	<i>exp.verilog_forcible()</i> <b>not</b> <i>exp.verilog_forcible()</i> <i>exp.verilog_forcible() == bool</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>bool</i> Either TRUE or FALSE. The default is FALSE.

By default, Verilog wires are not forcible. This Boolean attribute allows forcing of Verilog wires. The **verilog\_wire()** attribute (see [9.7.2.16](#)) needs to be specified before using this attribute.

*Syntax example:*

```
keep u.p.verilog_forcible()
```

#### 9.7.2.15 verilog\_strobe()

<b>Purpose</b>	Specify the sampling event for a Verilog object
<b>Category</b>	Verilog port attribute
<b>Syntax</b>	<i>exp.verilog_strobe() == string</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>string</i> A string specifying any legal Verilog timing control.

This string attribute specifies the sampling event for the Verilog signal that is bound to an external input port. This attribute is equivalent to the **verilog variable ... using strobe** declaration.

*Syntax example:*

```
keep u.p.verilog_strobe() == "@posedge clk1"
```

### 9.7.2.16 verilog\_wire()

<b>Purpose</b>	Create a single driver for a port (or multiple ports)
<b>Category</b>	Verilog port attribute
<b>Syntax</b>	<i>exp.verilog_wire()</i> <b>not</b> <i>exp.verilog_wire()</i> <i>exp.verilog_wire() == bool</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>bool</i> Either TRUE or FALSE. The default is FALSE.

This Boolean attribute allows an external out port to be bound to a Verilog wire. The main difference between this attribute and the **driver()** attribute is the **verilog\_wire()** attribute merges all of the ports containing this attribute into a single Verilog driver, while the **driver()** attribute creates a separate driver for each port.

*Syntax example:*

```
keep u.p.verilog_wire()
```

### 9.7.2.17 vhdl\_delay\_mode()

<b>Purpose</b>	Specify whether short pulses are propagated through the driver
<b>Category</b>	HDL port attribute
<b>Syntax</b>	<i>exp.vhdl_delay_mode() == mode-option</i>
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.
	<i>mode-option</i> Either <b>TRANSPORT</b> (the default) or <b>INERTIAL</b> .

This attribute specifies whether pulses whose period is shorter than the delay specified by the **driver\_delay()** attribute are propagated through the driver. **INERTIAL** specifies such pulses are not propagated, **TRANSPORT** that all pulses, regardless of length, are propagated.

This attribute also influences what happens if another driver (either VHDL or another unit) schedules a signal change, and before that change occurs, this driver schedules a different change. With **INERTIAL**, the first change never occurs.

This attribute is silently ignored, unless the **driver\_delay()** attribute is also specified.

*Syntax example:*

```
keep u.p.vhdl_delay_mode() == INERTIAL
```

## 9.8 Buffer port methods

The following methods are used to read from or write to buffer ports, and to check whether a buffer port queue is empty or full.

### 9.8.1 get()

<b>Purpose</b>	Read and remove data from an input buffer port queue
<b>Category</b>	Predefined TCM for buffer ports
<b>Syntax</b>	<i>in-port-instance-name</i> . <b>get()</b> : port element type
<b>Parameters</b>	<i>in-port-instance-name</i> An expression that returns an input buffer port instance.

Reads a data item from the buffer port queue and removes the item from the queue. Since buffer ports use a FIFO queue, **get()** returns the first item that was written to the port.

The thread blocks upon **get()** when there are no more items in the queue. If the queue is empty, or if it has a buffer size of 0 and no **put()** has been done on the port since the last **get()**, then the **get()** is blocked until a **put()** is done on the port.

The number of consecutive **get()** actions that is possible is limited to the number of items inserted by **put()**.

*Syntax example:*

```
rec_cell = in_port.get()
```

### 9.8.2 put()

<b>Purpose</b>	Write data to an output buffer port queue
<b>Category</b>	Predefined TCM for buffer ports
<b>Syntax</b>	<i>out-port-instance-name</i> . <b>put()</b> ( <i>data</i> : port element type)
<b>Parameters</b>	<i>out-port-instance-name</i> An expression that returns an output buffer port instance.
	<i>data</i> A data item of the port element type.

Writes a data item to the output buffer port queue. The sampling event of this TCM is **sys.any**. The new data item is placed in a FIFO queue in the output buffer port.

The thread blocks upon **put()** when there is no more room in the queue, i.e., when the number of consequent **put()** operations exceeds the **buffer\_size()** of the port instance. If the queue is full, or if it has a buffer size of 0 and no **get()** has been done on the port since the last **put()**, then the **put()** is blocked until a **get()** is done on the port.

The number of consecutive **put()** actions that is possible is limited to the buffer size.

*Syntax example:*

```
out_port.put(trans_cell)
```



### 9.8.3 is\_empty()

<b>Purpose</b>	Check if an output buffer port queue is empty
<b>Category</b>	Pseudo-method for buffer ports
<b>Syntax</b>	<i>in-port-instance-name.is_empty()</i> <b>not</b> <i>in-port-instance-name.is_empty()</i> <i>in-port-instance-name.is_empty() == bool</i>
<b>Parameters</b>	<i>in-port-instance-name</i> An expression that returns an input buffer port instance.

Returns TRUE if the input port queue is empty. Returns FALSE if the input port queue is not empty.

*Syntax example:*

```
var readable : bool;
readable = not cell_in.is_empty()
```

### 9.8.4 is\_full()

<b>Purpose</b>	Check if an output buffer port queue is full
<b>Category</b>	Pseudo-method for buffer ports
<b>Syntax</b>	<i>out-port-instance-name.is_full()</i> <b>not</b> <i>out-port-instance-name.is_full()</i> <i>out-port-instance-name.is_full() == bool</i>
<b>Parameters</b>	<i>out-port-instance-name</i> An expression that returns an output buffer port instance.

Returns TRUE if the output port queue is full. Returns FALSE if the output port queue is not full.

*Syntax example:*

```
var overflow : bool;
overflow = cell_out.is_full()
```

## 9.9 MVL methods for simple ports

The predefined port methods in this subclause are for reading and writing MVL data between ports, to facilitate communication with objects where MVL values occur. These methods operate on data of type mvl, which is defined as follows:

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

The enumeration literals are the same as those of VHDL, except for MVL\_N, which corresponds to the VHDL-("don't care") literal.

The MVL methods are applicable according to the port direction. Methods that write a value to a port are accessible for output and inout simple ports, while methods that read a value from a port are accessible for input and inout simple ports.

Accessing a port with MVL methods and accessing it through the \$ operator is allowed (*mixed access*).

### 9.9.1 MVL four-value logic

Some MVL methods operate on a subset of the enumeration in [9.9](#), MVL\_X, MVL\_Z, MVL\_0, and MVL\_1, which corresponds to the four-value logic of Verilog. To convert from nine-value logic to four-value logic, the mapping shown in [Table 23](#) is used.

**Table 23—MVL logic mapping**

Nine value	Four value
MVL_U, MVL_W, MVL_X, MVL_N	MVL_X
MVL_L, MVL_0	MVL_0
MVL_H, MVL_1	MVL_1
MVL_Z	MVL_Z

### 9.9.2 MVL string

Several functions allow specifying the MVL value or returning an MVL value expressed as string. A format of MVL string is the number of bits followed by the ' sign, the radix, and then the MVL literals. When an MVL list is converted into a string, the mapping shown in [Table 24](#) is used.

The mapping is done in the following way:

**Table 24—MVL string conversion**

MVL value	String
MVL_U	u
MVL_X	x
MVL_0	0
MVL_1	1
MVL_Z	z
MVL_W	w
MVL_L	L
MVL_H	h
MVL_N	n

When a string is converted to a list of mvl, the mapping is case-insensitive.

### 9.9.3 put\_mvl()

<b>Purpose</b>	Put an mvl data on a port of a non-mvl type	
<b>Category</b>	Predefined method for simple ports	
<b>Syntax</b>	<i>exp</i> . <b>put_mvl</b> ( <i>value</i> : mvl)	
<b>Parameters</b>	<i>exp</i>	An expression that returns a simple port instance.
	<i>value</i>	An mvl value.

Places an mvl value on an output or inout simple port, e.g., to initialize an object to a “disconnected” value. Placing an mvl value on a port whose element type is wider than one bit places the value in the LSB of the element.

*Syntax example:*

```
p.put_mvl(MVL_Z)
```

### 9.9.4 get\_mvl()

<b>Purpose</b>	Read mvl data from a port of a non-mvl type	
<b>Category</b>	Predefined method for simple ports	
<b>Syntax</b>	<i>exp</i> . <b>get_mvl</b> (): mvl	
<b>Parameters</b>	<i>exp</i>	An expression that returns a simple port instance.

Reads an mvl value from an input or inout simple port, e.g., to check that there are no undefined x bits. Getting an mvl value on a port whose element type is wider than one bit returns the value in the LSB of the element.

*Syntax example:*

```
check that pbi.get_mvl() != MVL_X else dut_error("Bad value")
```

### 9.9.5 put\_mvl\_list()

<b>Purpose</b>	Put a list of mvl values on a port of a non-mvl type	
<b>Category</b>	Predefined method for simple ports	
<b>Syntax</b>	<i>exp</i> . <b>put_mvl_list</b> ( <i>values</i> : list of mvl)	
<b>Parameters</b>	<i>exp</i>	An expression that returns a simple port instance.
	<i>values</i>	A list of mvl values.

Writes a list of mvl values to an output or inout simple port. Putting a list of mvl values on a port whose element type is a single bit writes only the LSB of the list.

*Syntax example:*

```
pbo.put_mvl_list({MVL_H; MVL_0; MVL_L; MVL_0})
```

### 9.9.6 get\_mvl\_list()

<b>Purpose</b>	Get a list of mvl values from a port of a non-mvl type
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> . <b>get_mvl_list()</b> : list of mvl
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.

Reads a list of mvl values from an input or inout simple port.

*Syntax example:*

```
check that not pbil.get_mvl_list().has(it == MVL_U)
else dut_error("Bad list")
```

### 9.9.7 put\_mvl\_string()

<b>Purpose</b>	Put an mvl value on a port of a non-mvl type when a value is represented as a string
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> . <b>put_mvl_string</b> ( <i>value</i> : string)
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.
	<i>value</i> An mvl value in the form of a base and one or more characters, entered as a string. The mvl values in the string shall be lowercase. Use 1 for MVL_1, 0 for MVL_0, z for MVL_Z, and so on.

Writes a string representing a list of mvl values to a simple output or inout port. See also [9.9.2](#).

*Syntax example:*

```
pbol.put_mvl_string("32'hxxxx1111")
```

### 9.9.8 get\_mvl\_string()

<b>Purpose</b>	Get a value in form of a string from a port of a non-mvl type
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> . <b>get_mvl_string</b> ( <i>radix</i> : radix): string
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.
	<i>radix</i> One of BIN, OCT, or HEX.

Returns a string in which each character represents an mvl value. See also [9.9.2](#).

*Syntax example:*

```
print pbis.get_mvl_string(BIN)
```

### 9.9.9 get\_mvl4()

<b>Purpose</b>	Get an mvl value from a port, converting nine-value logic to four-value logic
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> . <b>get_mvl4</b> (): mvl
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.

Reads a nine-value mvl value from an input simple port and converts it to four-value subset mvl. See also [9.9.1](#).

*Syntax example:*

```
check that pbi.get_mvl4() != MVL_Z else dut_error("Bad value")
```

### 9.9.10 get\_mvl4\_list()

<b>Purpose</b>	Get a list of mvl value from a port, converting nine-value logic to four-value logic
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> . <b>get_mvl4</b> (): list of mvl
<b>Parameters</b>	<i>exp</i> An expression that returns a simple port instance.

Reads a list of nine-value mvl values from an input simple port and converts them to four-value mvl. See also [9.9.1](#).

*Syntax example:*

```
check that not pbi4l.get_mvl4_list().has(it == MVL_X)
else dut_error("Bad list")
```

### 9.9.11 get\_mvl4\_string()

<b>Purpose</b>	Get a four-state value in form of a string from a port of a non-mvl type	
<b>Category</b>	Predefined method for simple ports	
<b>Syntax</b>	<i>exp</i> .get_mvl4_string( <i>radix</i> : radix): string	
<b>Parameters</b>	<i>exp</i>	An expression that returns a simple port instance.
	<i>radix</i>	One of BIN, OCT, or HEX.

Returns a string representing a four-value logic value. The mvl are first converted into four-value logic (see [9.9.1](#)) and then converted to a string (see [9.9.2](#)).

The returned string always includes all the bits, with no implicit extensions. For example, a port of type `int` returns a string of 32 characters, since an `int` is a 32-bit data type.

*Syntax example:*

```
print pbi4s.get_mvl4_string(BIN)
```

### 9.9.12 has\_x()

<b>Purpose</b>	Determine if a port has X	
<b>Category</b>	Predefined method for simple ports	
<b>Syntax</b>	<i>exp</i> .has_x(): bool	
<b>Parameters</b>	<i>exp</i>	An expression of a simple port type.

Returns `TRUE` if at least one bit of the port is `MVL_X`.

*Syntax example:*

```
print pbi4s.has_x()
```

### 9.9.13 has\_z()

<b>Purpose</b>	Determine if a port has Z	
<b>Category</b>	Predefined method for simple ports	
<b>Syntax</b>	<i>exp</i> .has_z(): bool	
<b>Parameters</b>	<i>exp</i>	An expression of a simple port type.

Returns `TRUE` if at least one bit of the port is `MVL_Z`.

*Syntax example:*

```
print pbi4s.has_z()
```

### 9.9.14 has\_unknown()

<b>Purpose</b>	Determine if a port has an unknown value
<b>Category</b>	Predefined method for simple ports
<b>Syntax</b>	<i>exp</i> . <b>has_unknown()</b> : bool
<b>Parameters</b>	<i>exp</i> An expression of a simple port type.

Returns TRUE if at least one bit of the port is one of the following: MVL\_U, MVL\_X, MVL\_Z, MVL\_W, or MVL\_N.

*Syntax example:*

```
print pbi4s.has_unknown( )
```

## 9.10 Global MVL routines

The subclause describes the global routines for manipulating MVL values.

### 9.10.1 string\_to\_mvl()

<b>Purpose</b>	Convert a string to a list of mvl values
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>string_to_mvl</b> ( <i>value-string</i> : string): list of mvl
<b>Parameters</b>	<i>value-string</i> A string representing mvl values.

Converts a string into a list of mvl (see [9.9.2](#)).

*Syntax example:*

```
mlist = string_to_mvl("8'bxz1")
```

### 9.10.2 mvl\_to\_string()

<b>Purpose</b>	Convert a list of mvl values to a string
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>mvl_to_string</b> ( <i>mvl-list</i> : list of mvl, <i>radix</i> : radix): string
<b>Parameters</b>	<i>mvl-list</i> A list of mvl values.
	<i>radix</i> One of BIN, OCT, or HEX.

Converts a list of mvl values to a string (see [9.9.2](#)). A sized number shall always be returned as a string.

*Syntax example:*

```
mstring = mvl_to_string({MVL_Z; MVL_Z; MVL_Z; MVL_Z;
                        MVL_X; MVL_X; MVL_X; MVL_X}, BIN)
```

### 9.10.3 mvl\_to\_int()

<b>Purpose</b>	Convert a list of mvl to an integer
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>mvl_to_int</b> ( <i>mvl-list</i> : list of mvl, <i>mask</i> : list of mvl): uint
<b>Parameters</b>	<i>mvl-list</i> A list of mvl values to convert to an integer value.
	<i>mask</i> A list of mvl values that are to be converted to 1.

Converts each value in a list of mvl values into a bit (1 or 0), using a list of mvl mask values to determine which mvl values are converted to 1.

When the list is less than 32 bits, it is padded with 0's. When it is greater than 32 bits, it is truncated, leaving the 32 least-significant bits.

Syntax example:

```
var ma : uint = mvl_to_int(1, {MVL_X})
```

### 9.10.4 int\_to\_mvl()

<b>Purpose</b>	Convert an integer value to a list of mvl values
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>int_to_mvl</b> ( <i>value</i> : uint, <i>mask</i> : mvl): list of mvl
<b>Parameters</b>	<i>value</i> An integer value to convert to a list of mvl values.
	<i>mask</i> An mvl value that replaces each bit in the integer that has the value 1.

Maps each bit that has the value 1 to the mask mvl value, retains the 0 bits as MVL\_0, and returns a list of 32 mvl values. The returned list always has a size of 32.

Syntax example:

```
var mlist : list of mvl = int_to_mvl(12, MVL_X)
```



### 9.10.5 mvl\_to\_bits()

<b>Purpose</b>	Convert a list of mvl values to a list of bits
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>mvl_to_bits</b> ( <i>mvl-list</i> : list of mvl, <i>mask</i> : list of mvl): list of bit
<b>Parameters</b>	<i>mvl-list</i> A list of mvl values to convert to bits.
	<i>mask</i> A list of mvl values that specifies which mvl values are to be converted to 1.

Converts a list of mvl values to a list of bits, using a mask of mvl values to indicate which mvl values are converted to 1 in the list of bits.

*Syntax example:*

```
var bl : list of bit = mvl_to_bits({MVL_Z; MVL_Z; MVL_X; MVL_L},
                                {MVL_Z; MVL_X})
```

### 9.10.6 bits\_to\_mvl()

<b>Purpose</b>	Convert a list of bits to a list of mvl values
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>bits_to_mvl</b> ( <i>bit-list</i> : list of bit, <i>mask</i> : mvl): list of mvl
<b>Parameters</b>	<i>bit-list</i> A list of bits to convert to mvl values.
	<i>mask</i> An mvl value that replaces each bit in the list that has the value 1.

Maps each bit with the value 1 to the mask mvl value, retains the 0 bits as MVL\_0, and returns an mvl list that has a size of *bit-list*.

*Syntax example:*

```
var ml : list of mvl = bits_to_mvl({1; 0; 1; 0}, MVL_Z)
```

### 9.10.7 mvl\_to\_mvl4()

<b>Purpose</b>	Convert an mvl value to a four-value logic value
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>mvl_to_mvl4</b> ( <i>value</i> : mvl): mvl
<b>Parameters</b>	<i>value</i> An mvl value to convert to a four-value logic value.

Converts an mvl value to a subset of four-value logic (see [9.9.1](#)).

*Syntax example:*

```
var m4 : mvl = mvl_to_mvl4(MVL_U)
```

#### 9.10.8 mvl\_list\_to\_mvl4\_list()

<b>Purpose</b>	Convert a list of mvl values to a list of four-value logic subset values
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>mvl_list_to_mvl4_list</b> ( <i>mvl-list</i> : list of mvl): list of mvl
<b>Parameters</b>	<i>mvl-list</i> A list of mvl values to convert to a list of four-value logic subset values.

Converts a list of mvl values to a list of the four-value logic subset (see [9.9.1](#)).

*Syntax example:*

```
var m4l : list of mvl = mvl_list_to_mvl4_list({MVL_N; MVL_L; MVL_H; MVL_1})
```

#### 9.10.9 string\_to\_mvl4()

<b>Purpose</b>	Convert a string to a list of four-value logic mvl subset values
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>string_to_mvl4</b> ( <i>value-string</i> : string): list of mvl
<b>Parameters</b>	<i>value-string</i> A string representing mvl values, consisting of a width and base followed by a series of characters corresponding to nine-value logic values.

Converts a string into a list of mvl, using the four-value logic subset. Logically, the string is converted to a list of mvl (see [9.9.2](#)), then converted into the four-value logic subset (see [9.9.1](#)).

*Syntax example:*

```
m1ist = string_to_mvl4("8'bxz")
```

### 9.11 Comparative analysis of ports and tick access

The *e* language supports both tick access (see [24.3](#)) and ports in order to access external simulated objects. Ports have the following advantages:

- They support modularity and encapsulation by explicitly declaring interfaces to *e* units.
- They are typed.
- They improve the performance of accessing DUT objects with configurable names.
- They can pass not only single values, but also other kinds of information, such as events and queues.
- They can be accompanied in *e* with generic or simulator-specific attributes that can be used to specify information needed for enhanced access to DUT objects.

*Example 1*

This example shows how tick access notation translates to MVL methods, assuming the following numeric port declaration:

```

data : inout simple_port of int is instance;
  keep bind(data, external);
  keep data.hdl_path() == "data";
d: int;

d = 'data';                      d = data$;

'data' = 32'bz;                  data.put_mvl_list(32'bz);

check that 'data@x' == 0;        check that not data.get_mvl_list().has
                                  (it == MVL_X);
                                  check that not data.has_x();

d = 'data[31:10]@z'              d = mvl_to_int(data.get_mvl_list(),
                                  {MVL_Z})[31:0]

```

*Example 2*

This example shows how tick access notation translates to use of an MVL port, assuming the following MVL port declaration:

```

data : inout simple_port of list of mvl is instance;
  keep bind (data, external);
  keep data.hdl_path() == "data";

check that 'data@x' == 0;        check that not data$.has(it == MVL_X);
                                  check that not data.has_x();

'data' = 32'bz                   data$ = 32'bz

```

## 9.12 *e* Port Binding

*e* ports can be bound imperatively by calling **do\_bind()** as well as declaratively using the bind syntax defined in **bind()** (9.7.2.1).

### 9.12.1 do\_bind()

<b>Purpose</b>	Connect ports	
<b>Category</b>	Predefined routine	
<b>Syntax</b>	<b>do_bind(port-exp1, port-exp2[,...]);</b> <b>do_bind(port-exp1, external);</b> <b>do_bind(port-exp1, empty   undefined);</b>	
<b>Parameters</b>	<i>port-exp1, port-exp2[,...]</i>	One or more expressions of port type. If two expressions are given and the port types are compatible, the two port instances are connected.
	<b>external</b>	Defines a port as connected to an external object.
	<b>empty</b>	Defines a disconnected port. Runtime accessing of a port with an empty binding is allowed.
	<b>undefined</b>	Defines a disconnected port. Runtime accessing of a port with an undefined binding causes an error

By calling the **do\_bind()** routine, you procedurally connect a port to one or more *e* ports or to one or more external simulated object. Ports can also be left explicitly disconnected with **empty** or **undefined**.

*Syntax example:*

```
do_bind(driver.bfm.data_in, bfm.driver.data_out)
```

### Notes

- The **do\_bind()** method can only be called during the **connect\_ports()** sub-phase. Calling it at any other time results in an error message.
- It is an error to declare a port disconnected in more than one way.

### 9.13 TLM Interface Ports in *e*

This clause defines the support the *e* language provides for interface ports, used to implement Transaction Level Modeling (TLM) standard interfaces. These ports facilitate the transfer of transactions between verification components, taking advantage of the standardized, high level TLM communication mechanism.

#### 9.13.1 interface\_port

<b>Purpose</b>	Transfer transactions between <i>e</i> units or a combination of <i>e</i> units and external modules	
<b>Category</b>	Unit member	
<b>Syntax</b>	<code>port-instance-name : [list of] direction <b>interface_port</b> of tlm-intf-type [<b>using prefix</b>=prefix   <b>using suffix</b>=suffix] [<b>is instance</b>]</code> <code>port-instance-name : [list of] <b>export interface_port</b> of tlm-intf-type [<b>is instance</b>]</code>	
<b>Parameters</b>	<i>port-instance-name</i>	A unique <i>e</i> identifier used to refer to the port or access its value.
	<i>direction</i>	in or out. There is no default.
	<i>tlm-intf-type</i>	<p>One of the supported TLM interface types specified in Table 25 or Table 26. The following restrictions apply to the “type” parameter of these interfaces.</p> <p>For internal <i>e</i> TLM interface ports, the type (or types) you specify for the interface can be any legal <i>e</i> type.</p> <p>External <i>e</i> TLM interface ports support transactions of a struct (or class) type only. Thus, for externally bound <i>e</i> TLM interface ports, the type (or types) you specify for the interface must be legal <i>e</i> types that inherit from any_struct.</p>
	<b>using prefix</b> =prefix <b>using suffix</b> =suffix	<p>Applies for <i>e</i> TLM input ports only. Specifies a prefix or suffix string to be attached to the predefined TLM methods for the given port.</p> <p>Using a prefix or suffix ensures that there are no method name collisions if a port contains more than instance of an <i>e</i> TLM interface port tied to the same TLM interface.</p> <p>(This syntax can be used only for the port instance members. It cannot be used in other declarations, such as declarations for parameters or variables.)</p>

An *e* TLM interface port type is parameterized with a specific TLM interface type. For example, if you define an *e* TLM interface port with the syntax `interface_port` of `tlm_nonblocking_put`, you have tied that port to the `tlm_nonblocking_put` interface. You can then use the set of methods (functions) predefined for that interface to exchange transactions.

*Syntax examples:*

```
e_packet : in interface_port of tlm_put of packet is instance;
```

```
p1 : out interface_port of tlm_nonblocking_transport of
(packet, msg) is instance;
```

```
p2 : export interface_port of tlm_blocking_put is
instance; // export
```

### 9.13.1.1 Special Port Types

#### 9.13.1.1.1 Export

An export interface port is a port whose enclosing unit does not implement the required interface methods. The interface methods are delegated to the connected unit. An export TLM input port in *e* is functionally equivalent to a SystemVerilog or SystemC export.

The following limitations apply to export interface ports:

- The port shall have an outbound connection.
- The port shall be connected (either directly or indirectly) to an input interface port or to an external port providing suitable interface functions.
- The port shall have no inbound connection.
- The port must be connected using the **connect()** (see [9.13.3.2.1](#)) method. The **bind()** constraints and the **do\_bind()** routine are not applicable for it.

#### 9.13.1.2 Analysis port

Analysis ports are ports featuring the `tlm_analysis` interface—a restricted write-only interface intended to share monitoring information for analysis purposes. They may have multiple outbound connections in support of broadcast implementations.

### 9.13.2 Defining Input *e* TLM interface ports

When a unit contains an instance member of an input TLM interface port, the unit must implement all methods required by the TLM interface type of that input port. The list of methods is predefined according to the standard TLM specification.

These methods must be defined before the port is defined. (If the methods and port are defined in the same module, however, the order does not matter.) If any of the required methods is missing, a compile time error shall be issued.

*Syntax example:*

```
struct packet {
    ...
};

unit server {
    // The following four lines define the four methods required
    // by the TLM interface tlm_put.
    put(value : packet)@sys.any is {...};
    try_put(value: packet) : bool is {...};
    can_put() : bool is {...};
    ok_to_put() : tlm_event is {...};
    packet_in : in interface_port of tlm_put of
                packet is instance;
}
```

In this example, the unit `server` implements the four methods/tasks which are required by the interface `tlm_put` of `packet`.

See below for a description of interface method semantics (see [9.13.4.5](#)).

### 9.13.3 Binding *e* TLM interface ports

#### 9.13.3.1 Binding Rules for TLM interface ports

A TLM output port can be bound to a TLM input port if the interface type of the output port is either the same as the interface type of the input port or subset of it (with exactly the same element type in the template parameter). For example, you can bind an output port of `tlm_nonblocking_put` to an input port of `tlm_put`, because the `tlm_nonblocking_put` interface is a subset of the `tlm_put` interface. Additionally:

- Empty and undefined bindings are supported for *e* TLM interface ports.
- Multiple binding is not supported for *e* TLM interface ports, except for analysis ports.
- Unification of ports bound to the same external port is not supported for *e* TLM interface ports.

#### 9.13.3.2 Declarative and Procedural Binding

*e* TLM interface ports have to be bound before usage, similar to any other port. Binding can be done declaratively with keep **bind()** constraints or procedurally with a **do\_bind()** or **do\_bind\_unit()** pseudo-routines.

*Syntax examples for declarative binding*

```
keep bind(port1, port2);
keep bind(port3, external)
```

*Syntax examples for procedural binding*

```
connect_ports() is also {
    do_bind(port1, port2);
    do_bind(port3, external)
}
```

##### 9.13.3.2.1 connect()—Language-Neutral Binding

External binding of TLM ports in a language-neutral way shall be supported by the simulation environment. The port method **connect()** is provided for this purpose. Use **connect()** when you want to bind two ports which are not both defined in the same language. For example, you can use this method to bind a SystemC port to a SystemVerilog port from *e*. For uniformity, **connect()** may be used to procedurally bind together *e* ports as well.

The **connect()** method shall be called once during the **connect\_ports()** phase. The effect of this method is immediate - it shall issue an error in case of any mismatch (wrong external path, mismatching interface types, unsupported multiple binding, and so on).

*Syntax of connect()*

```
<port1-exp>.connect(<port2-exp>);
<port1-exp>.connect(empty|undefined);
<port1-exp>.connect("external path")
```

*Syntax examples for connect()*

```
env.agent[1].my_port.connect(env.agent[2].my_export);
env.agent.monitor.port.connect(empty);
```

#### Description

The following restrictions shall apply to connections created by calling **connect()**.

If port1 is an output port:

- It can be connected to another *e* output port, *e* export port, or *e* input port.
- It can be connected to empty. In this case, this must be the only outbound connection it has. In this case, invoking a method on this port is like calling an empty method.
- It can be connected to undefined. In this case, this must be the only outbound connection it has. In this case, invoking a method on this port will cause a run time error.
- It can be connected to a specific external port by specifying the external port path. This external port can be of any direction.

If port1 is an export port:

- It can be connected to another *e* export port or to other *e* input port.
- It can be connected to a specific external port by specifying the external port path. This external port must be an input port or an export port.

Connecting to an external path:

- For SystemVerilog or SystemC, the external path must be quasi-static (full path from the top level scope).
- For *e*, the external path is an *e*-path, beginning with “sys”.

### 9.13.4 Supported TLM Interfaces

#### 9.13.4.1 tlm\_event Predefined Struct

The `tlm_event` predefined struct is used to synchronize a writer and a non-blocking reader on *e*-to-*e* TLM ports.

The predefined struct `tlm_event` is defined as follows:

```
struct tlm_event {
    event trigger;
    notify() is {
        emit trigger
    }
}
```

Some TLM functions return this struct. User code for an input port shall call `tlm_event.notify()` when it is ready to accept the next transaction. User code performing a non-blocking wait on that input shall be sensitive to emission of the event `tlm_event.trigger` and will write the next transaction when that event is emitted.

#### 9.13.4.2 Supported Unidirectional TLM Interfaces

NOTE—Non-blocking TLM interface calls are zero-delay calls. The blocking interface calls, which correspond to *e* TCM methods, consume an additional Specman tick and one cycle of simulation time.



**Table 25—Supported TLM Interfaces and Related Methods**

<b>TLM Interface</b>	<b>Interface Methods</b>
<b>Blocking Unidirectional Interfaces</b>	
tlm_blocking_put of type	put(value:type)@sys.any
tlm_blocking_get of type	get(value:*type)@sys.any
tlm_blocking_peek of type	peek(value:*type)@sys.any
tlm_blocking_get_peek of type	get(value:*type)@sys.any peek(value:*type)@sys.any
<b>Non-blocking Unidirectional Interfaces</b>	
tlm_nonblocking_put of type	try_put(value:type) : bool can_put() : bool ok_to_put() : tlm_event
tlm_nonblocking_get of type	try_get(value:*type) : bool can_get() : bool ok_to_get() : tlm_event
tlm_nonblocking_peek of type	try_peek(value:*type) : bool can_peek() : bool ok_to_peek() : tlm_event
tlm_nonblocking_get_peek of type	try_get(value:*type) : bool can_get() : bool ok_to_get() : tlm_event try_peek(value:*type) : bool can_peek() : bool ok_to_peek() : tlm_event
<b>Combined Unidirectional Interfaces (Blocking and Non-blocking)</b>	
tlm_put of type	put(value:type)@sys.any try_put(value:type) : bool can_put() : bool ok_to_put() : tlm_event
tlm_get of type	get(value:*type)@sys.any try_get(value:*type) : bool can_get() : bool ok_to_get() : tlm_event
tlm_peek of type	peek(value:*type)@sys.any try_peek(value:*type) : bool can_peek() : bool ok_to_peek() : tlm_event

### 9.13.4.3 Supported Bidirectional TLM Interfaces

NOTE—Non-blocking TLM interface calls are zero-delay calls. The blocking interface calls, which correspond to *e* TCM methods, consume an additional Specman tick and one cycle of simulation time.

**Table 26—Supported Bidirectional TLM Interfaces and Related Methods**

TLM Interface	Interface Methods
<b>Blocking Bidirectional Interfaces</b>	
tlm_blocking_master of (req-type, rsp-type)	put(value: req-type)@sys.any get(value: *rsp-type)@sys.any peek(value: *rsp-type)@sys.any
tlm_blocking_slave of (req-type, rsp-type)	put(value: rsp-type)@sys.any get(value: *req-type)@sys.any peek(value: *req-type)@sys.any
tlm_blocking_transport of (req-type, rsp-type)	transport(request: req-type, response: *rsp-type)@sys.any
<b>Non-blocking Bidirectional Interfaces</b>	
tlm_nonblocking_master of (req-type, rsp-type)	try_put(value: req-type) : bool can_put() : bool ok_to_put() : tlm_event try_get(value: *rsp-type): bool can_get(): bool ok_to_get(): tlm_event try_peek(value: *rsp-type): bool can_peek(): bool ok_to_peek(): tlm_event
tlm_nonblocking_slave of (req-type, rsp-type)	try_put(value: rsp-type) : bool can_put() : bool ok_to_put() : tlm_event try_get(value: *req-type): bool can_get(): bool ok_to_get(): tlm_event try_peek(value: *req-type): bool can_peek(): bool ok_to_peek(): tlm_event
tlm_nonblocking_transport of (req-type, rsp-type)	nb_transport(request: req-type, response: *rsp-type): bool
<b>Combined Bidirectional Interfaces (Blocking and Non-blocking)</b>	
tlm_master of (req-type, rsp-type)	put(value: req-type)@sys.any get(value: *rsp-type)@sys.any peek(value: *rsp-type)@sys.any try_put(value: req-type): bool can_put(): bool ok_to_put(): tlm_event try_get(value: *rsp-type): bool can_get(): bool ok_to_get(): tlm_event try_peek(value: *rsp-type): bool can_peek(): bool ok_to_peek(): tlm_event

**Table 26—Supported Bidirectional TLM Interfaces and Related Methods**

TLM Interface	Interface Methods
tlm_slave of (req-type, rsp-type)	put(value: rsp-type)@sys.any get(value: *req-type)@sys.any peek(value: *req-type)@sys.any try_put(value: rsp-type): bool can_put(): bool ok_to_put(): tlm_event try_get(value: *req-type): bool can_get(): bool ok_to_get(): tlm_event
lm_transport of (req-type, rsp-type)	transport(request: req-type, response: *rsp-type)@sys.any nb_transport(request: req-type, response: *rsp-type): bool

#### 9.13.4.4 Supported Analysis TLM Interface

**Table 27—Supported Analysis TLM Interface and Related Methods**

TLM Interface	Interface Methods
tlm_analysis of type	write(value : type)

#### 9.13.4.5 Required Semantics of TLM Interface Methods

TLM interface methods need to be implemented for each interface type. These methods are activated in response to a port interface call. As users of the *e* language define new interface types, they will have to provide implementations to the interface methods. The following section defines the expected semantics of the various Interface Methods.

##### 9.13.4.5.1 put(value:type)

The **put()** method passes on a value into the port, making it available for connected ports to read. This call shall block if the port is not ready to handle the transfer of the value.

##### 9.13.4.5.2 try\_put(value:type) : bool

The **try\_put()** method is non-blocking. If the port is ready to handle a put operation, the value is passed on and the method returns TRUE. Otherwise the method returns FALSE, and the value is not passed on.

##### 9.13.4.5.3 can\_put() : bool

The **can\_put()** method returns TRUE if the port is ready to handle a put operation (a call to put() will not block). FALSE is returned if the port is not ready to handle a put operation (a call to put() would block).

**9.13.4.5.4 ok\_to\_put() : tlm\_event**

The method **ok\_to\_put()** returns an event that will trigger each time the port is ready to handle a put operation. The returned event may be used to invoke the user code producing the next put operation.

**9.13.4.5.5 get(value:\*type)**

The **get()** method returns the value that is read from the port (the value is passed by reference in the parameter). This call blocks if no value is available to be read.

**9.13.4.5.6 try\_get(value:\*type) : bool**

The **try\_get()** non-blocking method returns TRUE and the read value if the port can be read. FALSE is returned if the port is not ready to be read (the **get()** operation would block).

**9.13.4.5.7 can\_get() : bool**

The method **can\_get()** returns TRUE if a get operation can be performed without blocking. FALSE is returned otherwise.

**9.13.4.5.8 ok\_to\_get() : tlm\_event**

The method **ok\_to\_get()** returns an event that will trigger each time the port is ready for a get operation (data is available for reading). The returned event can be used to trigger user code performing a get operation.

**9.13.4.5.9 peek(value:\*type)**

The **peek()** method returns the next value ready to be read from a port. The **peek()** method doesn't consume the value—a subsequent **get()** call will return the same value. The **peek()** method shall block if no value is ready to be read, and return only when the next value is available.

**9.13.4.5.10 try\_peek(value:\*type) : bool**

The **try\_peek()** non-blocking method returns TRUE and the read value if the port can be read. FALSE is returned if the port is not ready to be read (the **peek()** operation would block). This method doesn't consume the read value.

**9.13.4.5.11 can\_peek() : bool**

The method **can\_peek()** returns TRUE if a peek operation can be performed without blocking. FALSE is returned otherwise.

**9.13.4.5.12 ok\_to\_peek() : tlm\_event**

The **ok\_to\_peek()** method returns an event that triggers each time the port is ready for a peek operation (data is available to be read). The returned event can be used to trigger user code that monitors (performs a non-destructive inspection) the ports output.

**9.13.4.5.13 transport(request: req-type, response: \*rsp-type)**

The **transport()** method implements the equivalent of a procedure call, or a bidirectional atomic transfer. The first parameter contains the input to the procedure. The second parameter passes back the output (by reference). The method may consume time, depending on the user-level implementation of this method for a particular interface.

**9.13.4.5.14 write(value : type)**

The **write()** method passes on a value into an analysis port. The method is non-blocking, because analysis ports should always be ready to be written.



## 10. Constraints and generation

*Test generation* is a process producing data layouts according to a given specification. The specifications are provided in the form of *type declarations* and *constraints*. Constraints are statements that restrict values assigned to data items by test generation.

A constraint can be viewed as a property of a data item or as a relation between several data items. Therefore, it is natural to express constraints using Boolean expressions. Any valid Boolean expression in *e* can be turned into a constraint. Also, there are few special syntactic constructs not based on Boolean expressions for defining constraints.

Constraints can be applied to any data types including user-defined scalar types as well as struct and list types. It is natural to mix data types in one constraint, e.g.,

```
keep my_list.has(it == 0xff) => my_struct1 == my_struct2
```

### 10.1 Types of constraints

Constraints can be subdivided according to several criteria.

- a) Explicit or implicit
  - 1) *Explicit constraints* are those declared using the **keep** statement or inside **keeping {...}** block.
  - 2) *Implicit constraints* are those imposed by type definitions and variable declarations.

Implicit constraints are always hard.

*Examples*

```
x : int[1, 3, 5, 10..100];    \ is the same as:
```

```
x : int;
keep x in [1, 3, 5, 10..100];
```

```
l[20] : list of int;        \ is the same as:
```

```
l : list of int;
keep l.size() == 20
```

- b) Hard or soft
  - 1) *Hard constraints* are honored whenever the constrained data items are generated. A situation when a hard constraint contradicts other hard constraints, and thus cannot be honored, shall result in an error.
  - 2) *Soft constraints* are honored if they do not contradict hard constraints or soft constraints honored earlier. If a soft constraint cannot be honored, it is disregarded. (See [10.2.6](#) for the explanations on how the selection of soft constraints is done.)
- c) Simple or compound
 

A constraint combining other constraints in a Boolean combination using **not**, **and**, **or**, and **=>** is called *compound*. Otherwise, the constraint is called *simple*.

### 10.2 Generation concepts

This subclause describes the basic concepts of generation.

### 10.2.1 The basic flow of generation

Generation can be initiated for any field or variable. For items of struct types, the generation allocates the struct storage and recursively generates all generatable fields of the struct. All fields of a struct are considered generatable, except for the fields prefixed with ! (see [6.8](#)). There is no specific order in which data items or the fields in a struct hierarchy are generated.

For list items, the generation allocates the list and recursively generates all its elements. There is no specific ordering for whether list items are generated after the size of the list has been fixed or that the items are generated in the order of their indexes. Constraints specified for the items can impose restrictions on the list size or on the items specified earlier in the list.

For scalar types, such as `int`, `uint`, `bool`, etc., the generation only generates the respective value.

The following ordering rules, however, do apply:

- a) **pre\_generate()** and **post\_generate()**
  - 1) **pre\_generate()** of a struct is called after the struct is allocated and initialized using **init()**, but before any of the fields of the struct are generated. In particular, for a struct containing nested structs, the **pre\_generate()** method is called before any of the **pre\_generate()** methods of the nested structs.
  - 2) **post\_generate()** is called after the generation of all fields of the struct is finished. In particular, for a struct containing nested structs, the **post\_generate()** method is called only when all the nested generations are finished.

- b) **Methods**

A method accepting a generatable item as an argument is called after that item is fully generated.

#### Example

```
struct s {
    a : int;

    b : t;           // 't' is some other struct type

    keep a == f(b)
}
```

The constraint `a==f(b)` implies `b` is fully generated, including the calls to its **pre\_generate()** and **post\_generate()** before `f` is called on `b`. See also [10.2.2](#) and [10.2.3](#).

### 10.2.2 Using methods in constraints

Constraint paths can include method calls. The syntax is:

`[simple-path.]method-name([parameter, ...])[trailing-path]`

where *simple-path* does not include method calls and the following restrictions apply:

- If *simple-path* is generatable, then it is fully generated before the method is called.
- Generatable paths used as parameters of the method are fully generated before the method is called.
- For methods returning pointers to structs, the trailing path is sampled after evaluating the method and used as an input of the constraint.



*Example*

```

struct s {
    x : int[0..5];

    q : t;

    keep x < m(q).y;

    m(param:t): t is {
        result = param
    }
};

struct t {
    y : int[0..5]
}

```

In this example, *q* is generated before *x* and then *q* is used as an input in the constraint  $x < m(q).y$ . If *q.y* generates to 0, then the constraint  $x < m(q).y$  fails.

**10.2.2.1 Classification of methods**

Methods are classified into three categories.

- a) Methods that behave like mathematical functions (*pure*). The computed result is entirely determined by the arguments passed to the method. Multiple calls to the method with the same parameters always produce the same result.  
The use of such methods in constraints is safe and unrestricted.
- b) Methods that observe the “state of the world,” but do not change it. Such method can read fields, signals, global configuration flags, etc., and base the computation on that data. Multiple calls to the method with the same parameters can produce different results.

When using the methods of this category of constraints the following rules apply:

- 1) The method shall not base its computation on the items of the current generatable context, unless such items are passed as parameters to the method.

*Example*

```

struct packet {
    data      : list of byte;
    checksum  : uint;

    keep checksum == calc_checksum(data);

    calc_checksum(data:list of byte): uint is {
        // use 'data' to calculate checksum
    }
}

```

```

    }
}

```

This is correct; data is generated before the method is called.

- 2) The timing of the call and/or the number of calls to the method cannot be presumed, especially for methods reading values of the real-time or process clocks, operating-system (OS) environment variables, sizes of allocated memory, etc.

#### Example

```

extend sys {
  l[1000] : list of uint;

  keep for each in l {
    it == read_machine_real_time_clock_msec()
  }
}

```

It is incorrect to assume the method `read_machine_real_time_clock_msec` is called 1000 times, i.e., once for each list element in order (see [10.2.2.2](#)). It is acceptable for the generator to assume this method is a pure function, and thus, call it only once for the list and assign the result to all the list elements. It is also acceptable to assign values to list elements unrelated to their natural order of indexes. Thus, (normally in the presence of other constraints) the times read by the method might not be ordered with respect to the list indexes.

- c) Methods that observe and change the “state of the world.”

The use of such methods in constraints can create problems. Instead, use the corresponding operations within the **post\_generate()** method.

#### Example

```

struct packet {
  data : list of data_item;

  post_generate() is {
    var id;

    for each in data do {
      if it.x < 100 then {
        it.id = id;
        id += 1
      }
    }
  }
}

```

In general, it is impossible to classify methods automatically into the above three categories. Therefore, the following warnings shall be used if a method calling issue occurs:

*method call warning #1*: a method used in a constraint contains a non-local path anywhere in its body.

*method call warning #2*: a method used in a constraint contains an explicit assignment to a non-local path.

### 10.2.2.2 Number of calls

A method used in constraints can be called zero or more times. The number of calls to a method is irrelevant for the semantics of the constraint if the method behaves as a *pure* function [see [10.2.2.1](#), category [a](#)]. However, the results of generation can differ depending on the number of calls for the methods with side effects. Therefore, avoid using the methods of category [c](#), and only use methods of category [b](#) with caution.

### 10.2.3 Generatable paths and the sampling of inputs

The purpose of constraints is to constrain *generatable items*, i.e., those items that can be assigned random values (by the generator) satisfying the constraints. Thus, it is important to define which items are considered generatable and when.

In the context of the initial generation, all fields of **sys** and all fields of nested structs are generatable, except the fields declared as non-generatable (using the **!** prefix).

In the context of a **gen item** action (see [10.5.1](#)), *item* is generatable and, if *item* is of a struct type, all its nested fields are generatable—except the fields marked with **!**. If **gen item** action applies to a field defined as non-generatable, the *item* becomes generatable; however, any nested non-generatable fields remain non-generatable.

Example

```
struct packet {
    x : int;
    !y : int
};

extend sys {
    p1 : packet;    -- generated during pre-generation
    !p2 : packet;   -- skipped during pre-generation

    post_generate() is also {
        gen p2      -- this allocates p2 and generates p2.x but not p2.y
    }
}
```

Data items in constraints are referenced by using *paths* (see [4.3.4](#)). In generation context, each path is either generatable or non-generatable. *Generatable paths* refer to items that are assigned values during the

generation with respect to the corresponding constraints. Each constraint shall have all its inputs sampled before the items referenced by the generatable paths are generated.

Non-generatable paths refer to items that are not affected by generation, but those items might affect generatable items. Thus, non-generatable paths refer to *inputs* of constraints. A path is *non-generatable* if

- a) it is an absolute path (e.g., `sys.counter`).
- b) it includes method calls (e.g., `x.y.m( ).z`).
- c) it includes *do-not-gen* fields (e.g., `x.y.non_gen_field.z`).
- d) the path is **me** (e.g., `keep root_node => parent == me;`).

Otherwise, the path is generatable.

An otherwise generatable path can be defined as input to a constraint using the **value()** syntax, e.g., `keep x<value(y)`. In this case, the set of values *y* can take is unaffected by the constraints on *x*. The parameter *y* is treated as an input.

Arbitrary expressions can be used as arguments of **value()**. For example, in `keep x<value(y+z)`, both *y* and *z* become inputs of the constraint. The constraint resolution engine generates *y* and *z* (unaffected by the possible values of *x*) and computes their sum, which is then used as an input in the constraints.

Semantically, **value()** can be viewed as an identity function

**value**(*arg* : TYPE) is { *result* = *arg* }

defined for each type TYPE known to the generator. The use of **value()** in constraints is thus identical to the use of such an identity function.

A constraint that has no generatable paths with respect to the current generation context shall be reported as an error.

#### 10.2.4 Special cases of inputs in constraints

For some constraints, it is convenient to assume some of the parameters are always treated as inputs. One natural example is *method calls*. For a constraint, such as `keep x==f(y,z)`, *y* and *z* are presumed to be generated first and then their values are used as inputs in the context of `x==f(y,z)`. If *x* cannot accept the value returned by the call to `f(y,z)`, the generation results in a contradiction error. Thus, given the values of arguments, the constraint resolution engine is presumed to be able to compute and assign the value of the method/function call. The converse is not presumed, however.

There are four kinds of constraints treating some of their parameters as inputs, even if these parameters represent generatable paths.

- a) *method calls*: all arguments of a method call are treated as inputs.
- b) *bit slice*: the two boundaries *i* and *j* of a bit slice, such as in `keep x[j:i]==y`, are treated as inputs. It means that *i* and *j* are generated first and their values used as inputs for solving it with respect to *x* and *y*. Thus, the constraint resolution engine is not required to deduce the values of the bit slice boundaries. Namely,

`keep 0b101010010101[j:i] == 0b100`

is allowed to cause a contradiction error.

- c) *list segments*: in an expression `l[i..j]`, the segment boundaries *i* and *j* are treated as inputs in the generation of *l*. Thus, a constraint such as

```
keep ({1; 2; 3; 4; 5})[i..j] in {2; 3}
```

is allowed to cause a contradiction error.

- d) *shift operations*: in expressions such as  $x < k$  and  $x > k$ , the number of bits for shifting is treated as input. Thus, a constraint such as

```
keep 0b1110011010 >> k == 0b1110
```

is allowed to cause a contradiction error.

### 10.2.5 The scope of constraints

A constraint can be either applicable or inapplicable depending on the context of generation. There are two basic rules governing that aspect of generation.

- a) All constraints defined for **sys** and any of the nested structs are applicable during the initial generation.
- b) For generation started by the **gen item** action (see [10.5.1](#)), the following are applicable:
  - 1) The constraints defined within the optional *constraints* block.
  - 2) All constraints defined in the type of *item*, if *item* is of a struct type.
  - 3) All constraints referring to *item* in this struct (**me**) and in the struct hierarchy containing **me**.

*Example*

```
struct packet {
    x : uint;
    y : uint;
    keep x < y
};

extend sys {
    !p1 : packet;
    keep p1.y == 8;
    !p2 : packet;

    post_generate() is also {
        gen p1 keeping {it.x > 5};
        p2 = new;
        gen p2.x
    }
}
```

The generation of *p1* succeeds. The applicable constraints here are  $p1.x > 5$  (by rule b1),  $p1.x < p1.y$  (by rule b2), and  $p1.y == 8$  (by rule b3) Thus,  $p1.y$  becomes 8 and  $p1.x$  becomes either 6 or 7.

The generation of `p2.x` fails. For `p2` allocated using `new`, `p2.x=0` and `p2.y=0`. The only applicable constraints in this case is `p2.x < p2.y` (by rule b3). `p2.y` is not a generatable item here in the context of `gen p2.x` (see [10.2.3](#)); it is used as input, so the constraint is equivalent to `p2.x < 0`. Since `x` is a **uint**, the constraint is not satisfiable.

### 10.2.6 Soft constraints

A constraint can be declared as soft by prefixing it with the **soft** keyword in the declaration. See also [10.4.4](#).

```
keep soft constraint;

gen item keeping {soft constraint; ...};

keep soft item = select {...}
```

Intuitively, soft constraints are satisfied if possible and otherwise disregarded. Soft constraints suggest default values and relations that can be overridden by hard or other soft constraints. They are considered with respect to the order of importance, which is a reverse of the (textual) order of soft constraints in the model.

The following properties of soft constraints also apply:

- Assume two soft constraints  $c_1$  and  $c_2$ , such that  $c_1$  is more important than  $c_2$ . Then the generator shall always produce a solution satisfying  $c_1$ , if one exists. It is also required that the generator find a solution satisfying both  $c_1$  and  $c_2$ , if it exists.
- Assume a collection of data items (fields and/or variables)  $x_1 \dots x_n$ , a collection of constraints  $c_1 \dots c_k$  linking the data items, and a solution exists satisfying all  $c_1 \dots c_k$ . Then a solution needs to be found for `{soft  $c_1$ ; ... ; soft  $c_k$ }` such that all soft constraints are satisfied.

Informally, this property means that in the absence of hard constraints, soft constraints act as hard, except for those cases causing contradictions.

#### Example

```
struct s {
    x : int;
    y : int;
    z : int;
    keep x in [1..100];
    keep x < y or y < z
}
```

is the same as

```
struct s {
    x : int;
    y : int;
    z : int;
    keep soft x in [1..100];
    keep soft x < y or y < z
}
```

```

}

```

#### 10.2.6.1 keep gen-item.reset\_soft()

<b>Purpose</b>	Quit evaluation of soft constraints for a field
<b>Category</b>	Struct member
<b>Syntax</b>	<b>keep</b> <i>gen-item.reset_soft()</i>
<b>Parameters</b>	<i>gen-item</i> A generatable item (see <a href="#">10.4.11</a> ).

This causes the program to quit the evaluation of soft value constraints for the specified field. Soft constraints for other fields are still evaluated. Soft constraints are considered in reverse order to the order in which they are defined in the *e* code.

The syntax **keep** *gen-item.reset\_soft()* is used for discarding soft constraints referring to the *gen-item* loaded so far. Soft constraints not referring to *gen-item* or soft constraints referring to *gen-item*, but loaded later, are taken into account by the constraint resolution engine. The main use of this feature is for overloading the default “soft” behavior of a model.

Syntax example:

```

keep c.reset_soft()

```

### 10.2.6.2 keep soft... select

<b>Purpose</b>	Constrain distribution of values
<b>Category</b>	Struct member
<b>Syntax</b>	<b>keep soft</b> <i>gen-item</i> <b>==select</b> { <i>weight</i> : <i>value</i> ; ...}
<b>Parameters</b>	<i>gen-item</i> A generatable item of type list (see <a href="#">10.4.11</a> ).
	<i>weight</i> Any <b>uint</b> expression. Weights are proportions; they do not have to add up to 100. A relatively higher weight indicates a greater probability that the value is chosen.
	<i>value</i> <i>value</i> is one of the following: a) <b>range-list</b> —A range list such as [ 2 . . 7 ]. A select expression with a range list selects the portion of the current range that intersects with the specified range list. b) <b>exp</b> —A constant expression. A select expression with a constant expression (usually a single number) selects that number, if it is part of the current range. c) <b>others</b> —Selects the portions of the current range that do not intersect with other select expressions in this constraint. Using a weight of 0 for <b>others</b> causes the constraint to be ignored, i.e., the effect is the same as if the <b>others</b> option were not entered at all. d) <b>pass</b> —Ignores this constraint and keeps the current range as is. e) <b>edges</b> —Selects the values at the extreme ends of the current range(s). f) <b>min</b> —Selects the minimum value of the <i>gen-item</i> . g) <b>max</b> —Selects the maximum value of the <i>gen-item</i> .

This specifies the relative probability that a particular value or set of values is chosen from the current range of legal values. The current range is the range of values as reduced by hard constraints and by soft constraints that have already been applied. A weighted value shall be assigned with the probability of

$$\text{weight}/(\text{sum of all weights})$$

Weights are treated as integers. If an expression is used for a *weight*, the value of the expression shall be smaller than the maximum integer size (MAX\_INT).

Like other soft constraints, **keep soft select** is order dependent (see [10.2.6](#)) and shall not be met if it conflicts with hard constraints or soft constraints that have already been applied. In those cases where some values conflict with other constraints, **keep soft select** shall bias the distribution based on the remaining permissible values.

Syntax example:

```
keep soft me.opcode == select {
    30 : ADD;
    20 : ADDI;
    10 : [SUB, SUBI]
}
```



## 10.2.7 Constraining non-scalar data types

This subclause describes constraining structs and lists.

### 10.2.7.1 Constraining structs

There are two basic constraints that apply to structs: struct equality and struct inequality. Other constraints affecting items of struct types (such as list constraints with structs as list elements) can be equivalently expressed using these basic constraints and Boolean combinators.

#### 10.2.7.1.1 Struct equality

*Struct equality* constraints two structs to share the same struct layout, i.e., it *aliases* two struct pointers.

*Example*

```
struct packet {
    x : int;
    y : int
};

extend sys {
    p1 : packet;
    p2 : packet;
    keep p1 == p2;

    post_generate() is also {
        p1.x = 5
    }
}
```

This causes `p1` and `p2` to represent the same struct, i.e., `sys.p1` and `sys.p2` can be viewed as pointers pointing to the same place in memory. Thus, the assignment in `post_generate` has the same effect on both structures, i.e., `sys.p1.x = sys.p2.x = 5`.

In contrast,

```
struct packet {
    x : int;
    y : int
};

extend sys {
```

```

    p1 : packet;

    p2 : packet;

    keep p1.x == p2.x;

    keep p1.y == p2.y;

    post_generate() is also {

        p1.x = 5

    }

}

```

The first two lines in “extend sys” define two structures with the same contents, `sys.p1` and `sys.p2`. Then the assignment in `post_generate` changes the value of `sys.p1.x`, but not of `sys.p2.x`. Thus, at the end `sys.p1.x=5`, while `sys.p2.x` is set to a random value from the range `[MIN_INT..MAX_INT]`. Of course, this value could be 5 as well, but the chance for that is  $1/(2^{32})$ . Thus, most likely at the end `sys.p1.x != sys.p2.x`.

#### 10.2.7.1.2 Struct inequality

*Struct inequality* states that two struct pointers cannot be aliased, although they can still have the identical contents. Normally, struct inequality only makes sense for structs with a finite set of possible values (see [10.2.7.2](#)).

##### Example

```

struct packet {

    x : int;

    y : int

};

extend sys {

    p1 : packet;

    p2 : packet;

    keep p1 in sys.list_of_input_packets;

    keep p2 in sys.list_of_input_packets;

    keep p1 != p2;

    keep p1.x == p2.x;

    keep p1.y == p2.y

}

```

This code constraints both `p1` and `p2` to be elements of a (pre-built) list of input packets, such that `p1` and `p2` are distinct packets and have the same contents. The generation succeeds if and only if (**iff**) the list `sys.list_of_input_packets` contains repetitions. There is no contradiction in the fact `p1` and `p2` are different structs with identical contents.

### 10.2.7.2 Allocation vs. aliasing

By default, a new structure is allocated for each item of a struct type. The only exception to that are the cases when the range of possible structs is limited by constraints to a finite number of choices.

*Example*

```
p : packet;

keep (packet == sys.input_packet1) or (packet == sys.input_packet2)
```

In this example, the range of values for `packet` is limited by the values `sys.input_packet1` and `sys.input_packet2`, where both values are pre-built structures, i.e., inputs to the constraint. In contrast,

```
keep packet != sys.input_packet1
```

does not limit the choices of `packet` to a finite set. Here, there are an infinite number of ways to allocate `packet` so that it does not point to `sys.input_packet1`. Thus, the system allocates a NEW struct for `packet` in this case. This behavior makes struct inequality redundant for those cases where the set of potential struct values is unlimited.

### 10.2.7.3 Constraining lists

This subclause describes constraining lists. See also [Table 25](#).

#### 10.2.7.3.1 List equality

*List equality constraint* states that two lists contain the same elements in the same order.

*Example*

```
extend sys {

  l1 : list of int;

  l2 : list of int;

  !x : int;

  keep l1 == l2;

  post_generate() is also {
    x = l2.pop()
  }
}
```

This generates two identical lists `l1` and `l2`. Then, `post_generate()` removes the last element of `l2` and preserves it in `x`. `l1` and `l2` are not aliased to the same list by the list equality constraint, they are “copies.” Therefore, `l2.pop()` does not remove the last value of `l1`.

#### 10.2.7.3.2 List inequality

The list inequality constraint (`l1 != l2`) states that the items of list type `l1` and `l2` are different. Namely:

- a) The number of elements in the lists is different; or
- b) The number of elements is the same and there is an index  $i$  such that

$$l1[i] \neq l2[i]$$

### 10.2.7.3.3 List item

The syntax *generatable\_path\_to\_list[index]* provides a generatable path of a list element. This syntax can be used in constraints as any other generatable path. List item constraints are fully solvable. Thus, the constraint can be used in several different modes.

#### Examples

```
keep sys.packets[5] == x;      -- element extraction from fixed list
keep l[7] < 25;                -- constraining certain element of list
keep sys.packets[i].id == 10;  -- index look-up for fixed list and value
keep l[i] < x                  -- multi-way constraint
```

### 10.2.7.3.4 Item in list

The expression *item in list* states that *item* is an element of the *list*. Note that a constraint such as

```
keep x in l
```

also implies that *l* includes at least one element, i.e., it is non-empty.

### 10.2.7.3.5 List in list

The syntax *list1 in list2* provides the way of constraining two lists *list1* and *list2* so *list1* is a (possibly permuted) *sublist* of *list2*. *list1* is a possibly permuted sublist of *list2* if for every valid index  $i$  in *list1* there exists a matching valid index  $j$  in *list2* such that  $list1[i] == list2[j]$ . Every index  $j$  of *list2* is represented at most once in *list1*.

Informally, this definition means *list1* can be obtained from *list2* by a number (possibly zero) of **delete** operations of elements of *list2* and then applying **is\_a\_permutation(list2)**.

#### Examples

```
{1;2;3} is a sublist of {0;1;3;2;3}
{1;2;3} is a permuted sublist of {1;3;2}
{1;1;2} is a sublist of {1;3;1;4;2}
{1;1;2} is NOT a sublist of {1;2;2;3}
{1;1;2} is a permuted sublist of {2;1;1}
```

### 10.2.7.3.6 Permutations

The syntax *list1.is\_a\_permutation(list2)* states that *list1* is a permutation of *list2*. The lists *list1* and *list2* contain exactly the same elements and the same numbers of repetitions of each element.

#### Examples

```
{2;3;1} is a permutation of {1;2;3}
```

$\{2;3\}$  is not a permutation of  $\{1;2;3\}$   
 $\{1;2;3\}$  is a permutation of  $\{1;2;3\}$   
 $\{2;3;2;1\}$  is NOT a permutation of  $\{1;2;3\}$

**is\_a\_permutation** is a symmetric property, i.e., *list1* is a permutation of *list2* **iff** *list2* is a permutation of *list1*. Thus, the following two constraints are equivalent:

```

keep list1.is_a_permutation(list2);

keep list2.is_a_permutation(list1)

```

### 10.2.7.3.7 List attributes

There are several properties of lists that can be constrained using the *attribute* syntax, **list.attribute(...)**.

**list.size()**—constrains the size of the list, e.g., `keep my_list.size() in [5..8]`  
*my\_list* can have 5,6,7, or 8 elements.

**list.count(*exp*)**—counts the number of list elements satisfying *exp* that have a Boolean type, e.g.,  
`keep my_list.count(it == 3) == 5`  
the number 3 appears exactly five times in *my\_list*.

**list.has(*exp*)**—verifies at least one item of the list satisfies the Boolean *exp*. This is the same as  
`list.count(exp) > 0`.

**list.unique(*exp*)**—constrains the elements satisfying the Boolean *exp* so they are unique within the list, e.g., `keep my_list.unique(it.is a(RED packet))`  
ensures there are no duplicate RED packets in *my\_list*.

**list.sum(*exp*)**—constrains the sum of the list elements satisfying *exp* containing a Boolean type. The attribute applies only to lists of numeric type, e.g., `keep my_list.sum(it) == 100`  
for the elements of *my\_list* in the range [0..20] is 100.

### 10.2.7.3.8 Constraining all list items: keep for each

<b>Purpose</b>	Constrain list items
<b>Category</b>	Struct member
<b>Syntax</b>	<b>keep for each</b> [( <i>item-name</i> )] [ <b>using</b> [ <b>index</b> ( <i>index-name</i> )] [ <b>prev</b> ( <i>prev-name</i> )]] <b>in</b> <i>gen-item</i> [ <b>do</b> ] {( <i>constraint-bool-exp</i>   <i>nested-for-each</i> ); ...}
<b>Parameters</b>	<i>item-name</i> An optional name used as a local variable referring to the current item in the list. The default is <b>it</b> .
	<i>index-name</i> An optional name referring to the index of the current item in the list. The default is <b>index</b> .
	<i>prev-name</i> An optional name referring to the previous item in the list. The default is <b>prev</b> .
	<i>gen-item</i> A generatable item of type list (see <a href="#">10.4.11</a> ).
	<i>constraint-bool-exp</i> A simple or a compound Boolean expression (see <a href="#">10.4.10</a> ).
	<i>nested-for-each</i> A nested <b>for each</b> block, with the same syntax as the enclosing <b>for each</b> block, except that <b>keep</b> is omitted.

This defines a value constraint on multiple list items. The following restrictions also apply:

- **for each** constraints can be nested. The parameters *item-name*, *index-name*, and *prev-name* of a nested **for each** can shadow the names used in the outer **for each** blocks. In particular, if the optional names are unspecified, then the default names **it**, **index**, and **prev** refer to the corresponding details of the innermost **for each** block.
- Within a **for each** constraint, **index** represents a running index in the list, which is treated as a constant with respect to each list item.
- Generated items need to be referenced by using a pathname that starts either with **it**, **prev**, or the optional *item-name* or *prev-name*, respectively. Items whose pathname does not start with **it** can only be sampled; their generated values cannot be constrained.
- If a **for each** constraint is contained in a **gen ... keeping** action, the iterated variable needs to be named first.

Syntax example:

```
keep for each (p) in pkl do {
    soft p.protocol in [atm, eth]
}
```

### 10.2.7.3.9 All solutions

This feature generates lists of structs covering all possible combinations of values for certain fields. The syntax is *list.is\_all\_iterations(fieldname, ...)*, where *list* is a list of elements and *fieldname, ...* are field names of some struct type T. The arguments of **is\_all\_iterations** are unique, i.e., there are no repetitions in the list of fields. All fields shall be defined under the base type T, i.e., fields defined in **when** subtypes or **like** successors are not allowed.

*Example*

```

struct s {
    b1 : bool;
    b2 : bool;
    x  : int
};

extend sys {
    l : list of s;
    keep l.is_all_iterations(.b1, .b2)
}

```

The resulting `sys.l` includes four elements for all four combinations of TRUE/FALSE of `b1` and `b2`. The values of `x` are chosen randomly.

### 10.3 Type constraints

This subclause describes how to use type constraints to restrict the declared type of a field to one of its **like** or **when** subtypes for a given context. A constraint prefixed with the **type** modifier is both (a) enforced by the generator (like a regular constraint) and (b) presupposed at compile time for purposes of type checking. Expressions for which type constraints apply are automatically downcast to the specified subtype wherever required. This saves explicit downcasting ("`is_a()`" and "`as a`" operators) for the expression and lets the downcast expression be used as a generatable term (rather than input) in constraint contexts.

### 10.3.1 keep type

<b>Purpose</b>	Refine the type of a field to one of its subtypes for the specified context
<b>Category</b>	Struct member
<b>Syntax</b>	<pre> <b>keep type</b> [<b>me.</b>]field-name <b>is a</b> type <b>keep type</b> [<b>me.</b>]field-name.property-name == [<b>me.</b>]my-property-name <b>keep for each</b> [(item-name)] <b>in</b> list-field-name {     ...     <b>type</b> item-name <b>is a</b> type;     ... } <b>keep for each</b> [(item-name)] <b>in</b> list-field-name {     ...     <b>type</b> item-name.property-name == [<b>me.</b>]my-property-name;     ... } </pre>
<b>Parameters</b>	<i>field-name</i> The name of a struct field in the enclosing struct.
	<i>type</i> The name of a struct or unit type.
	<i>property-name</i> The name of an enumerated or Boolean const field.
	<i>my-property-name</i> The name of a field of the same type as the <i>property-name</i> in this constraint.
	<i>item-name</i> An optional name used as a local variable referring to the current item in the list. The default is <b>it</b> .
	<i>list-field-name</i> The name of a field of typelist of struct (or unit) in the enclosing struct.

You can put a type constraint either on a field of a struct type or on a list field of a struct type. The declaration is similar to a regular constraint inside a **keep** struct member, or, in the list case, inside a **keep for each** construct, with the **type** keyword prefixing the expression.

The **type** keyword is a constraint modifier syntactically analogous to **soft**. However, unlike **soft**, it can modify only specific constraint expressions and can appear only in restricted contexts.

The type correlation can be fixed or, when the correlated types are **when** subtypes, variable. The former case is expressed using the **is a** operator. In the latter case the determinant property (the **when** determinant) of the referenced struct is equated to a determinant property of the same type in the declaring struct type.

Type constraints affect the static semantics of field-access expressions of the form *instance-expression.field-name* (field-access in which *instance-expression* is omitted is equivalent to one having **me** as the *instance-expression*). Typically the static type of a field-access expression is determined according to the type of the field as it was initially declared in the struct type of *instance-expression* (or in one of its supertypes). Type constraints tying the static type of *instance-expression* with a subtype of the field's declared type can change this rule. If the context in which the field-access occurs requires the subtype, the field-access is automatically downcast. In this case, a runtime check is added to ensure that the casting is justified, and an error is issued if it is not. The runtime check involves a minor overhead, not more than that required by the **as\_a()** operator.



## NOTE—

- In the Boolean expression following **type**, operators other than == and **is a** are not allowed. For example, the following is not allowed:
 

```
keep type TRUE => engine is a FORD engine // not allowed
```
- The **for each** clause must occur immediately after **keep**. For example, the following is not allowed:
 

```
keep my_doors.size() > 4 => for each in my_doors { // not allowed
    type it is a small door
  }
```
- Type constraints can equate only constant fields, so the **const** keyword must appear in the declaration of fields involved in equality constraints.
- Type constraints in general affect code from that point onwards. This includes type constraints that appear inside a **for each** clause, in which case other expressions in the same scope after the declaration (but not before it) can assume automatic casting.
- Type constraints cannot appear inside a **gen** action.
- The **soft** keyword cannot be used with type constraints.
- As with non-type constraints, the determinant field of the when subtype is assigned only during generation. Thus the **pre\_generate()** method of the type specified in the type constraint is not called during generation.
- A field's type may be restricted by more than one type constraint with respect to different “when” dimensions (determinant fields).

## Syntax Example

```
keep type f.pl == pl;
keep for each in lf {
    type it is a B S1
};
```

**10.3.2 Type constraints and struct fields**

Automatic casting of a struct-reference field is performed in any context that requires it, including:

- Struct-member access
- Assignment
- Parameter passing

**10.3.3 Type constraints and list fields**

When the type relation is one-to-many, in other words, when a list field is concerned, automatic casting is applied not to the list itself but to its elements. Automatic casting affects list operators whose result type is the element type, such as indexing (the [] operator) and **pop()**. It also affects the iteration variable inside the **for each** construct, both in procedural and in constraint contexts.

**10.3.4 Type constraints and like subtypes**

Type constraints work just as well for **like** subtypes of the declared type of the field. They apply to the two “**is a**” forms of the **keep type** struct member.

Note that type constraints with **like** subtypes cannot make the actual **like** type of a generated field dependent on a **when** determinant. In other words, they may not figure under a **when** subtype if they affect a field not declared in the same subtype. This is an error: the constraint is unenforceable.

## 10.4 Defining constraints

This subclause describes the constructs used to define constraints. See also [4.10](#).

### 10.4.1 keep

<b>Purpose</b>	Define a hard value constraint
<b>Category</b>	Struct member
<b>Syntax</b>	<b>keep</b> <i>constraint-bool-exp</i>
<b>Parameters</b>	<i>constraint-bool-exp</i> A simple or a compound Boolean expression (see <a href="#">10.4.10</a> ).

This states restrictions on the values generated for fields in the struct or its subtree, or describes required relationships between field values and other items in the struct or its subtree.

Hard constraints are applied whenever the enclosing struct is generated. For any **keep** constraint in a generated struct, the generator either meets the constraint or issues a constraint contradiction message. If the **keep** constraint appears under a **when** construct, the constraint is considered only if the **when** condition is true.

Syntax example:

```
keep kind != tx or len == 16
```

### 10.4.2 keep all of {...}

<b>Purpose</b>	Define a constraint block
<b>Category</b>	Struct member
<b>Syntax</b>	<b>keep all of</b> { <i>constraint-bool-exp</i> ; ...}
<b>Parameters</b>	<i>constraint-bool-exp</i> A simple or a compound Boolean expression (see <a href="#">10.4.10</a> ).

A **keep** constraint block is exactly equivalent to a **keep** constraint for each constraint Boolean expression in the block. The **all of** block can be used as a constraint Boolean expression itself.

Syntax example:

```
keep all of {
    kind != tx;
    len == 16
}
```

### 10.4.3 keep struct-list.is\_all\_iterations()

<b>Purpose</b>	Cause a list of structs to have all iterations of a field	
<b>Category</b>	Constraint-specific list method	
<b>Syntax</b>	<b>keep</b> <i>gen-item.is_all_iterations</i> ( <i>field-name</i> : exp, ...)	
<b>Parameters</b>	<i>gen-item</i>	A generatable item of type list of struct (see <a href="#">10.4.11</a> ).
	<i>field-name</i>	The name of a scalar field of a struct. The field name shall be prefixed by a period ( . ). The order of fields in this list does not affect the order in which they are iterated. The specified field that is defined first in the struct is the one that is iterated first.

This causes a list of structs to have all legal, non-contradicting iterations of the fields specified in the field list. Fields not included in the field list are not iterated; their values can be constrained by other relevant constraints. The highest value always occupies the last element in the list.

Soft constraints on fields specified in the field list are skipped. All other relevant hard constraints on the list and on the struct are applied. If these constraints reduce the ranges of some of the fields in the field list, then the generated list is also reduced.

The following restrictions also apply:

- The number of iterations in a list produced by *list.is\_all\_iterations*() is the product of the number of possible values in each field in the list. Use the **absolute\_max\_list\_size** generation configuration option to set the maximum number of iterations allowed in a list (the default is 524 288).
- The *list.is\_all\_iterations*() method can only be used in a constraint Boolean expression.
- The fields to be iterated shall be of a scalar type, not a list or struct type.

Syntax example:

```
keep packets.is_all_iterations(.kind, .protocol)
```

### 10.4.4 keep soft

<b>Purpose</b>	Define a soft value constraint	
<b>Category</b>	Struct member	
<b>Syntax</b>	<b>keep soft</b> <i>constraint-bool-exp</i>	
<b>Parameters</b>	<i>constraint-bool-exp</i>	A simple Boolean expression (see <a href="#">10.4.10</a> ).

This suggests default values for fields or variables in the struct or its subtree, or describes suggested relationships between field values and other items in the struct or its subtree. The following restrictions apply:

- Soft constraints are order dependent (see [10.2.6](#)) and shall not be met if they conflict with hard constraints or soft constraints that have already been applied.
- The **soft** keyword cannot be used in compound Boolean expressions.

- Individual constraints inside a constraint block can be soft constraints.
- Because soft constraints only suggest default values, it is better not to use them to define architectural constraints.

Syntax example:

```
keep soft legal
```

#### 10.4.5 keep gen ... before

<b>Purpose</b>	Modify the generation order
<b>Category</b>	Struct member
<b>Syntax</b>	<b>keep gen</b> ( <i>gen-item</i> : exp, ...) <b>before</b> ( <i>gen-item</i> : exp, ...)
<b>Parameters</b>	<i>gen-item</i> An expression that returns a generatable item. The parentheses ( ( ) ) are required. See also <a href="#">10.4.11</a> .

This requires the generatable items specified in the first list to be generated before the items specified in the second list. This constraint can be used to influence the distribution of values by preventing soft value constraints from being consistently skipped (see [10.2](#)). The following restrictions also apply:

- This constraint itself can cause constraint cycles. If a constraint cycle involving one of the fields in the **keep gen ... before** constraint exists and if the **resolve\_cycles** generation configuration option is TRUE, the constraint can be ignored if the program cannot satisfy both it and other constraints that conflict with it.
- This constraint cannot appear on the LHS of an implication operator ( $=>$ ).

Syntax example:

```
keep gen (y) before (x)
```

#### 10.4.6 keep soft gen ... before

<b>Purpose</b>	Suggest order of generation
<b>Category</b>	Struct member
<b>Syntax</b>	<b>keep soft gen</b> ( <i>gen-item</i> : exp, ...) <b>before</b> ( <i>gen-item</i> : exp, ...)
<b>Parameters</b>	<i>gen-item</i> An expression that returns a generatable item. The parentheses ( ( ) ) are required. See also <a href="#">10.4.11</a> .

This modifies the *soft* generation order by recommending the fields specified in the first field list be generated before the fields specified in the second field list. This soft generation order is second in priority to the hard generation order created by dependencies between parameters and **keep gen before** constraints.

This constraint can be used to suggest a generation order that is later overridden in individual tests with a hard order constraint. This constraint cannot appear on the LHS of an implication operator ( $=>$ ).

Syntax example:

```
keep soft gen (y) before (x)
```

#### 10.4.7 keep gen\_before\_subtypes()

<b>Purpose</b>	Specify a <b>when</b> determinant field for deferred generation
<b>Category</b>	Struct member
<b>Syntax</b>	<b>keep gen_before_subtypes</b> ( <i>determinant-field</i> : field, ...)
<b>Parameters</b>	<p><i>determinant-field</i> An expression that evaluates to the name of a field in the struct type. The field shall have at least one value that is used as a <b>when</b> determinant for a subtype definition. If the field is not a <b>when</b> determinant field, a warning is issued and the constraint is ignored.</p> <p>Multiple field expressions can be entered, separated by commas ( , ).</p>

To speed up generation of structs with multiple **when** subtypes, this type of constraint, called a *subtype optimization constraint*, causes the generator engine to wait until a **when** determinant value is generated for a specified field before it analyzes constraints and generates fields under the **when** subtype.

When no subtype optimization constraints are present in a struct, the generator analyzes all of the constraints and fields in the struct before it generates the struct, even those constraints and fields that are defined under **when** subtypes. When a subtype optimization constraint is present, the generator initially analyzes only the constraints and fields of the base struct type. When a subtype optimization **when** determinant is encountered, the generator analyzes the associated **when** subtype and then generates it.

The following considerations also apply:

- Subtype optimization can handle multiple determinants. Subtypes are analyzed and generated in the order in which their **when** determinants are encountered.
- If multiple determinants are specified, and some of them are subtype optimization determinants while others are not, then a subtype that is a result of multiple inheritance of a subtype optimization determinant and a non-subtype optimization determinant shall be treated the same.
- The generator engine's ability to resolve contradictions is diminished somewhat by subtype optimization constraints. Specifically, the generator might not be able to resolve contradictions arising from constraints under subtypes that involve fields of the base type.
- The analysis and generation is recursive. If a subtype contains another determinant that is specified in a subtype optimization constraint, then that sub-subtype is analyzed and generated as soon as its determinant field is generated under the higher-level subtype.

Syntax example:

```
keep gen_before_subtypes (format)
```

#### 10.4.8 keep reset\_gen\_before\_subtypes()

<b>Purpose</b>	Disable all previous <b>keep gen_before_subtypes()</b> subtype optimization constraints
<b>Category</b>	Struct member
<b>Syntax</b>	<b>keep reset_gen_before_subtypes()</b>

When subtype optimization is turned off by default, this constraint causes the generator to ignore all previously defined **gen\_before\_subtypes()** constraints for the enclosing struct or unit. Any such constraints defined after the reset shall be followed.

When subtype optimization is turned on by default, this constraint turns off subtype optimization for the enclosing struct or unit. When subtype optimization is forced on or off, this constraint has no effect.

Syntax example:

```
keep reset_gen_before_subtypes( )
```

#### 10.4.9 value()

<b>Purpose</b>	Modify generation sequence
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<b>value</b> ( <i>item</i> : exp)
<b>Parameters</b>	<i>item</i> A legal <i>e</i> expression.

This generates values for any data items that are contained in the expression and returns the value of the expression. This method affects generation order and also makes the constraint unidirectional.

*Example*

```
keep a == value(b + c)
```

This constraint has two results.

- b and c are generated before a.
- The value of a cannot otherwise be constrained.

Syntax example:

```
keep i < value(j)
```

### 10.4.10 constraint-bool-exp

<b>Purpose</b>	Define a constraint on a generatable item
<b>Category</b>	Expression
<b>Syntax</b>	<i>bool-exp</i> [ <b>or</b>   <b>and</b>   <b>=&gt;</b> <i>bool-exp</i> ] ...
<b>Parameters</b>	<i>bool-exp</i> An expression that returns either TRUE or FALSE when evaluated at runtime.

A *constraint Boolean expression* is a simple or compound Boolean expression that describes the legal values for at least one generatable item or constrains the relation of one generatable item with others. A compound Boolean expression is composed of two or more simple expressions joined with the **or**, **and**, or implication (**=>**) operators. [Table 25](#) shows the *e* special constructs that are useful in constraint Boolean expressions.

**Table 25—Constraining Boolean expression**

Constraint	Definition
<b>soft</b>	A keyword that indicates the constraint is either a soft value constraint or a soft order constraint. See <a href="#">10.4</a> for a definition of these types of constraints.
<b>soft...select</b>	An expression that constrains the distribution of values.
<b>.reset_soft()</b>	A pseudo-method that causes the test generator to quit evaluation of soft constraints for a field, in effect, removing previously defined soft constraints.
<b>.is_all_iterations()</b>	A list method used only within constraint Boolean expressions that causes a list of structs to have all legal, non-contradicting iterations of the specified fields.
<b>.is_a_permutation()</b>	A list method that can be used within constraint Boolean expressions to constrain a list to have the same elements as another list.
<b>[not] in</b>	An operator that can be used within constraint Boolean expressions to constrain an item to a range of values or a list to be a subset of another list; or when used with <b>not</b> , to be outside the range or absent from another list.
<b>is [not] a</b>	An operator that checks the subtype of a struct.

The following considerations also apply:

- The **soft** keyword can be used in simple Boolean expressions, but not in compound Boolean expressions.
- The order of precedence for Boolean operators is: **and**, **or**, **=>**. A compound expression containing multiple Boolean operators of equal precedence is evaluated from left to right, unless parentheses ( ( ) ) are used to indicate expressions of higher precedence.
- Any *e* operator can be used in a constraint Boolean expression. However, certain operators can affect generation order or can create an constraint that is not enforceable.
- In compound expressions where multiple implication operators are used, the order in which the operations are performed is significant. For example, in the following constraint, the first expression (a **=>** b) is evaluated first by default:

```
keep a => b => c;           // is equivalent to:
```

```
keep (not a or b) => c;           // is equivalent to:
keep a and (not b) or c
```

However, adding parentheses around the expression (b => c) causes it to be evaluated first, with very different results.

```
keep a => (b => c);              // is equivalent to:
keep a => (not b) or c;          // is equivalent to:
keep (not a) or (not b) or c
```

### Examples

The following are examples of simple constraint Boolean expressions:

```
not short           // where "short" is of type "bool"
long == TRUE
soft x > y
x + z == y + 7
```

The following are examples of compound constraint Boolean expressions:

```
x > 0 and soft x < y
is_a_good_match(x, y) => z < 1024
color != red or resolution in [900..999]
packet is a good packet => length in [0..1023]
```

See also [5.1.1](#).

Syntax example:

```
z == x + y
```

#### 10.4.11 gen-item

<b>Purpose</b>	Identifies a generatable item
<b>Category</b>	Expression
<b>Syntax</b>	<code>[<b>me</b>.]field1-name[,field2-name ...]</code> <code>  <b>it</b>   [<b>it</b>.]field1-name[,field2-name ...]</code>
<b>Parameters</b>	<i>field-name</i> The name of a field in the current struct or struct type.

A *generatable item* is an operand in a Boolean expression that describes the legal values for that generatable item or constrains its relation with another generatable item. Every constraint shall have at least one generatable item or an error shall be issued.

In a **keep** constraint, the syntax for specifying a generatable item is a path starting with **me** of the struct containing the constraint and ending with a field name. In a **gen** action, the syntax for specifying a generatable item is a path starting with **it** of the struct containing the constraint and ending with a field name.



A generatable item cannot have an indexed reference in it, except as the last item in the path. See also [4.3.3](#).

Syntax example:

```
me.protocol
```

## 10.5 Invoking generation

There are two ways of invoking generation, as follows:

- a) Generation is invoked automatically when generating the tree of structures starting at **sys**.
- b) Generation can be called for any data item by using the **gen** action. The scope of this type of generation is restricted (see [10.5.1](#)). The generation order is (recursively):
  - 1) Allocate the new struct
  - 2) Call **pre\_generate()**
  - 3) Perform generation
  - 4) Call **post\_generate()**

### 10.5.1 gen

<b>Purpose</b>	Generate values for an item
<b>Category</b>	Action
<b>Syntax</b>	<b>gen</b> <i>gen-item</i> [ <b>keeping</b> {[ <i>it</i> ]. <i>constraint-bool-exp</i> ; ...}]
<b>Parameters</b>	<i>gen-item</i> A generatable item. If the expression is a struct, it is automatically allocated, and all fields under it are generated recursively, in depth-first order.
	<i>constraint-bool-exp</i> A simple or compound Boolean expression (see <a href="#">10.4.10</a> ).

This generates a random value for the instance of the item specified in the expression and stores the value in that instance, while considering all the constraints specified in the **keeping** block, as well as other relevant constraints at the current scope on that item or its children. Constraints defined at a higher scope than the enclosing struct are not considered.

The following considerations also apply:

- Values for particular struct instances, fields, or variables can be generated during simulation (on-the-fly generation) by using the **gen** action.
- This constraint can also be used to specify constraints that apply only to one instance of the item.
- The **soft** keyword can be used in the list of constraints within a **gen** action.
- The earliest the **gen** action can be called is from a struct's **pre\_generate()** method.
- The generatable item for the **gen** action cannot include an index reference.
- If a **gen ... keeping** action contains a **for each** constraint, the iterated variable needs to be named.

Syntax example:

```
gen next_packet keeping {
    .kind in [normal, control]
```

```
}
```

### 10.5.2 pre\_generate()

<b>Purpose</b>	Method run before generation of struct
<b>Category</b>	Method of <b>any_struct</b>
<b>Syntax</b>	<i>[struct-exp.]pre_generate()</i>
<b>Parameters</b>	<i>struct-exp</i> An expression that returns a struct. The default is the current struct.

The **pre\_generate()** method is run automatically after an instance of the enclosing struct is allocated, but before generation is performed. This method is initially empty, but can be extended to apply values procedurally to prepare constraints for generation. It can also be used to simplify constraint expressions before they are analyzed by the constraint resolution engine.

NOTE—Prefix the ! character (see 6.8) to the name of any field whose value is determined by **pre\_generate()**. Otherwise, normal generation overwrites this value.

Syntax example:

```
pre_generate() is also {
    m = 7
}
```

### 10.5.3 post\_generate()

<b>Purpose</b>	Method run after generation of struct
<b>Category</b>	Predefined method of <b>any_struct</b>
<b>Syntax</b>	<i>[struct-exp.]post_generate()</i>
<b>Parameters</b>	<i>struct-exp</i> An expression that returns a struct. The default is the current struct.

The **post\_generate()** method is run automatically after an instance of the enclosing struct is allocated and both pre-generation and generation have been performed. This method can be extended for **any\_struct** to manipulate values produced during generation. It can also be used to derive more complex expressions or values from the generated values.

Syntax example:

```
post_generate() is also {
    m = m1 + 1
}
```

## 11. Events

The *e* language provides temporal constructs for specifying and verifying behavior over time. All *e* temporal language features depend on the occurrence of events, which are used to synchronize activity with a simulator and within the *e* program.

### 11.1 Causes of events

[Table 26](#) describes how an event is made to occur.

**Table 26—Event causation**

Syntax	Cause of the event
<b>event a is</b> (@b and @c)@d	Derived from other events (see <a href="#">Clause 12</a> ).
<b>event a is rise</b> ('top.b')@sim	Derived from behavior of a simulated device (see <a href="#">Clause 12</a> ).
<b>event a;</b> meth_b()@c <b>is</b> { ... ; <b>emit</b> a; ... };	By the <b>emit</b> action in procedural code (see <a href="#">11.3.2</a> ).

### 11.2 Scope of events

Events are defined as a part of a struct definition. When a struct is instantiated, each instance has its own event instances. Each event instance has its own schedule of occurrences. There is no relation between occurrences of event instances of the same type. All references to events are to event instances.

The scoping rules for events are similar to other struct members, such as *fields*.

- If a path is provided, use the event defined in the struct instance pointed to by the path.
- If no path is provided, the event is resolved at compile time. The current struct instance is searched.
- If the event instance is not found, a compile-time error shall be issued.

### 11.3 Defining and emitting named events

This subclause describes the **event** and **emit** constructs.

### 11.3.1 event

<b>Purpose</b>	Define a named event	
<b>Category</b>	Struct member	
<b>Syntax</b>	<b>event</b> <i>event-type</i> [ <b>is</b> [ <b>only</b> ] <i>temporal-expression</i> ]	
<b>Parameters</b>	<i>event-type</i>	The name of the event type (any legal <i>e</i> identifier).
	<i>temporal-expression</i>	An event or combination of events and temporal operators. See also <a href="#">Clause 12</a> .

Events can be attached to temporal expressions (TEs), using the option **is** [**only**] *temporal-expression* syntax, or they can be unattached. An *attached event* is emitted automatically during any tick in which the TE attached to it succeeds. For a definition of the success of a TE, see [12.3](#).

Events, like methods, can be redefined in struct extensions. The **is only** *temporal-expression* syntax in a struct extension is used to change the definition of an event, e.g., to define an event once and then attach it to several different TEs under different **when** struct subtypes.

### 11.3.2 emit

<b>Purpose</b>	Cause a named event to occur	
<b>Category</b>	Action	
<b>Syntax</b>	<b>emit</b> [ <i>struct-exp</i> .] <i>event-type</i>	
<b>Parameters</b>	<i>struct-exp</i>	An expression referring to the struct instance in which the event is defined.
	<i>event-type</i>	The type of event to emit.

This causes an event of the specified type to occur.

- Emitting an event causes the immediate evaluation of all TEs containing that event.
- The **emit** event does not consume time. It can be used in regular methods and in TCMs.
- The simplest usage of **emit** is to synchronize two TCMs, where one TCM waits for the named event and the other TCM emits it.

Syntax example:

```
emit ready
```

## 11.4 Predefined events

Predefined events are emitted at particular points in time.

### 11.4.1 General predefined events

[Table 27](#) lists the general predefined events.

**Table 27—Predefined events**

Predefined event	Description
<b>sys.any</b>	Emitted on every tick.
<b>sys.tick_start</b>	Emitted at the start of every tick.
<b>sys.tick_end</b>	Emitted at the end of every tick.
<b>session.start_of_test</b>	Emitted once at test start.
<b>session.end_of_test</b>	Emitted once at test end.
<b>struct.quit</b>	Emitted when a struct's <b>quit()</b> method is called. Only exists in structs that contain events or have members that consume time (for example, TCMs or <b>on</b> struct members).
<b>sys.new_time</b>	In stand-alone operation (no simulator), this event is emitted on every <b>sys.any</b> event. When a simulator is being used, this event is emitted whenever a callback occurs and the attached simulator's time has changed since the previous callback.

#### 11.4.1.1 **sys.any**

This event is a special event that defines the finest granularity of time. The occurrence of any event in the system causes an occurrence of the **any** event at the same tick. In stand-alone *e* program operation (that is, with no simulator attached), the **sys.any** event is the only one that occurs automatically. It typically is used as the clock for stand-alone operation.

#### 11.4.1.2 **sys.tick\_start**

This event is provided mainly for visualizing and debugging the program flow.

#### 11.4.1.3 **sys.tick\_end**

This event is provided mainly for visualizing and debugging the program flow. It also can be used to provide visibility into changes of values that are computed during the tick, such as the values of coverage items.

#### 11.4.1.4 **session.start\_of\_test**

The first action the predefined **run()** method executes is to emit the **session.start\_of\_test** event. This event is typically used to anchor TEs to the beginning of a test.

#### 11.4.1.5 **session.end\_of\_test**

This event is typically used to sample data at the end of the test. This event cannot be used in TEs, as it is emitted after evaluation of TE has been stopped. The **on session.end\_of\_test** struct member is typically used to prepare the data sampled at the end of the test.

#### 11.4.1.6 **struct.quit**

This only exists in structs that contain temporal members (events, **on**, **expect**, or TCMs). It is emitted when the struct's **quit()** method is called, to signal the end of time for the struct.

The first action executed during the check test phase is to emit the **quit** event for each struct that contains it. This event can be used to cause the evaluation of TEs that contain the **eventually** temporal operator (and check for **eventually** TEs that have not been satisfied).

#### 11.4.1.7 sys.new\_time

This event is emitted on every **sys.any** event in stand-alone operation (no simulator). When a simulator is being used, this event is emitted whenever a callback occurs and the attached simulator's time has changed since the previous callback.

#### 11.4.2 Simulation time and ticks

Using any of the following expressions causes the DUT to be monitored for a change in that expression:

- **rise** | **fall** | **change** (*HDL expression*) @sim
- **wait delay** *expression*
- Verilog event

For each simulation delta cycle where a change in at least one of these monitored expressions occurs, the simulator passes control to the *e* program. If simulation time has advanced since the last time control was passed to the *e* program, a **new\_time** event is issued. In any case, **tick\_start** and **any** events are issued. Then, after emitting all events initiated by changes in monitored expressions in that simulation delta cycle, a **tick\_end** event is issued.

Thus, the **new\_time** event corresponds to a new simulation time slot and a tick corresponds to a simulation delta cycle where at least one monitored expression changes.

Multiple ticks can occur in the same simulation time slot under the following conditions:

- When a new value is driven into the DUT and that value causes a change in a monitored HDL object, as in a clock generator
- When a monitored event is derived from another monitored event, as in a clock tree
- When a zero-delay HDL subprogram is called from *e*

For an explanation of when values are assigned, see [19.5](#).

NOTE—Glitches that occur in a single simulation time slot are ignored; only the first occurrence of a particular monitored event in a single simulation time slot is recognized.

## 12. Temporal expressions

*Temporal expressions* (TEs) provide a declarative way to describe temporal behavior. The *e* language provides a set of temporal operators and keywords that can be used to construct TEs. A TE is a combination of events and temporal operators that describes behavior. A TE expresses temporal relationships between events, values of fields, variables, or other items during a test.

TEs are used to define the occurrence of events, specify sequences of events as checkers, and suspend execution of a thread until the given sequence of events occurs. TEs are only legal in the following constructs:

- **wait** and **sync** actions in TCMs
- **event** definitions and **expect** or **assume** struct members

### 12.1 Overview

TEs are built from a set of temporal atoms (see [12.1.2](#)) and a set of operators (see [12.2](#)).

TEs are interpreted over finite paths, which are successions of states. A *state* is a complete valuation of all atoms, i.e., for each atom, a given state tells us whether that atom holds or does not hold. A *path* is an abstraction of the execution of an *e* program. The notion of a state does away with the detail between the occurrence of **tick\_start** and a subsequent **tick\_end** when simulating the *e* program (see [11.4.2](#)). This detail is not needed to define the semantics of TEs.

The meaning of a TE is given in terms of tight satisfaction (“holds tightly”), which is a relationship between paths and TEs. A TE *holds tightly* on a path **iff** (if and only if) that complete path exhibits the behavior expressed by the TE.

- a) The first step to determine the meaning of a TE is to determine its sampling event (see [12.1.3](#)) and to transform it to its sampled normal form (see [12.1.4](#)).
- b) The second step is to apply the definitions of the operators (see [12.2](#)).
- c) New events can be defined in terms of TEs using the **event** construct (see [11.3.1](#)). It is not necessary to consider an atom’s origin when defining the semantics of TEs.
- d) To determine the success or failure of a TE, which is relevant for describing the meaning of constructs that use TEs, see [12.3](#).

#### 12.1.1 Terminology

This subclause defines some of the terms used in this clause.

##### 12.1.1.1 holds

A term used to talk about the meaning of atoms. If the atom is an event, the atom holds if the event occurs at the *state*. If the atom is a proposition `true(exp)`, the *proposition* holds if *exp* evaluates at the *state* to TRUE if it is a Boolean expression or to non-zero if *exp* is an numeric expression.

##### 12.1.1.2 holds tightly

A term used to talk about the meaning of TEs. A TE is interpreted over finite paths. Informally, a TE holds tightly on a path **iff** the path exhibits the behavior described by the TE.

### 12.1.1.3 occurs, occurrence

A term used to talk about the meaning of events. For each event, a *state* identifies whether the event is present or not.

### 12.1.1.4 path

A succession of *states* of the *e* program. For a path  $p$ , the notation  $p = s_0 s_1 \dots s_n$  can be used to indicate  $p$  has  $n+1$  states. The first state of  $p$  is  $s_0$ ; the last state of  $p$  is  $s_n$ .

### 12.1.1.5 prefix

A prefix of a path is a *sub-path* of that consists of all *states* of the original *path* up to a certain point. The empty path and the original path are both prefixes of the original path.

### 12.1.1.6 proposition

A *temporal atom* of the form  $\text{true}(\text{exp})$ .

### 12.1.1.7 sampling event

An atom associated with a TE that determines when the TE is evaluated. Every TE has a sampling event.

### 12.1.1.8 sampled normal form

The result of propagating the *sampling event* of a TE. In this form, every atom under the TE is explicitly sampled.

### 12.1.1.9 state

A valuation of all atoms. For each event, the state identifies whether the event occurs. For each *proposition*, the state identifies whether the proposition holds.

### 12.1.1.10 strict prefix

A prefix of the path that is not that original path.

### 12.1.1.11 sub-path

A contiguous part of a *path*. The sub-paths of a path  $p = s_0 s_1 \dots s_n$  are the paths  $sp = s_i s_{i+1} \dots s_j$ , where  $0 \leq i \leq j \leq n$ .

### 12.1.1.12 success

A term used to talk about the meaning of *top-level temporal expressions*. The TE succeeds at a point of a *path* if there is a *sub-path* ending in the given point, such that the TE *holds tightly* over the *sub-path*.

### 12.1.1.13 temporal atom

An event, a proposition  $\text{true}(\text{exp})$ , or the atom cycle.

### 12.1.1.14 tight satisfaction

See [12.1.1.2](#).



### 12.1.1.15 top-level temporal expression

A TE that is not nested underneath another TE, i.e., it appears directly under a **wait** or **sync** action, or an **event**, **expect**, or **assume** struct member.

### 12.1.2 Temporal atoms

TEs are built from a set of temporal atoms. These atoms consist of the following:

- Events, including predefined events, such as the event `sys.any`
- Propositions `true(exp)`
- The atom `cycle`

Atoms are interpreted at states of the *e* program. Each and every state identifies which atoms hold and which do not hold.

Events are said to “occur,” propositions “hold.” The proposition `true(exp)` holds at a state if *exp* evaluates to TRUE if it is a Boolean expression or to non-zero if *exp* is an numeric expression. The atom `cycle` holds at any state.

### 12.1.3 Sampling event

A key step in determining the meaning of a TE is to identify its sampling event. The sampling event for a TE is one of the following, in decreasing order of precedence:

- a) The sampling event specified with the binary @ operator.
- b) The sampling event inherited from the parent TE.
- c) The sampling event of the TCM if the TE appears under a **wait** or **sync** action of that TCM.
- d) `sys.any`, if none of the above applies.

### 12.1.4 Sampling propagation and sampled normal form

The first step in uncovering the meaning of a top-level TE is to propagate its sampling event so every temporal atom that appears under the given TE is explicitly sampled. This transformation is called *sampling propagation* and the result is the *sampled normal form* of the given TE.

The rules for transforming a TE to its sampled normal form are as follows:

- a) Given a (top-level) TE *t* whose sampling event is *q*,  
let *S* be a function that returns the sampled normal *S(t, q)*.
- b) Let *b* be an expression; *e* and *q* be events; *t*, *t1*, and *t2* be TEs;  
*exp* be a numeric expression; and *range* be either *exp.exp*, *exp..*, *..exp*, or *..*

$$S(\text{true}(b), q) = \text{true}(b) @q$$

$$S(\text{cycle}, q) = \text{cycle} @q$$

$$S(@e, q) = @e @q$$

$$S((t), q) = (S(t, q))$$

$$S(t1 \text{ and } t2, q) = S(t1, q) \text{ and } S(t2, q)$$

$S(t_1 \text{ or } t_2,$	$q) = S(t_1, q) \text{ or } S(t_2, q)$
$S(\{t_1 ; t_2\},$	$q) = \{S(t_1, q) ; S(t_2, q)\}$
$S([exp]*t,$	$q) = [exp]*S(t, q)$
$S(\sim[range]*t,$	$q) = \sim[range]*S(t, q)$
$S(\{[range]*t_1; t_2\}$	$q) = \{[range]*S(t_1, q); S(t_2; q)\}$
$S(t @ e,$	$q) = S(t, e) @q$
$S(\text{fail } t,$	$q) = \text{fail } (S(t, q))$
$S(t_1 \Rightarrow t_2,$	$q) = S(t_1, q) \Rightarrow S(t_2, q)$

$S(\text{rise}(exp),$	$q) = \text{rise}(exp) @q$
$S(\text{fall}(exp),$	$q) = \text{fall}(exp) @q$
$S(\text{change}(exp),$	$q) = \text{change}(exp) @q$
$S(\text{not } t,$	$q) = \text{not } (S(t, q))$
$S(\text{detach } (t),$	$q) = \text{detach } (S(t, q))$
$S(t \text{ exec action},$	$q) = S(t, q) \text{ exec action}$

### Example

Consider the following TE:

```
{@a; ~[...]*cycle; @b} @q
```

The sampling event of this TE is specified explicitly as  $q$ .

The temporal atoms  $a$ ,  $cycle$ , and  $b$  are not explicitly sampled.

To obtain the sampled normal form,  $q$  is propagated into the following sub-expressions:

```
{@a @q; ~[...]*cycle @q; @b @q} @q
```

## 12.2 Temporal operators and constructs

This subclause gives the semantics of the operators for building TEs. These definitions only apply after a top-level TE has first been interpreted to determine its sampling event and sampled normal form (see [12.1.4](#)). Each meaning is given in terms of tight satisfaction (see [12.1.1](#)).

To illustrate tight satisfaction, consider the TE  $cycle @q$ , which holds tightly on any path where event  $q$  occurs in the last state of that path and in no other state. It is clear that tight satisfaction considers the path as a whole and makes requirements on every state of the given path. This is different from the notion success of a TE (see [12.3](#)). The textual description of the operators may illustrate the meaning of the operators for a top-level expression using the notion of success (see [12.3](#)) rather than tight satisfaction.

### 12.2.1 Precedence of temporal operators

[Table 28](#) shows the precedence of temporal operators, listed from highest precedence to lowest.

**Table 28—Precedence of temporal operators**

Operator name	Operator example
<i>named event</i>	@ <i>event-name</i>
<b>exec</b>	TE <i>exec action-block</i>
<b>repeat</b>	[ ] * TE
<b>fail</b> <b>not</b>	fail TE not TE
<b>and</b>	TE1 and TE2
<b>or</b>	TE1 or TE2
<b>sequence</b>	{TE1 ; TE2}
<b>yield</b>	TE1 => TE2
<i>sample event</i>	TE @ <i>event-name</i>

### 12.2.2 cycle

<b>Purpose</b>	Specify an occurrence of a sampling event
<b>Category</b>	TE
<b>Syntax</b>	<b>cycle</b>
<b>Informal semantics</b>	<i>cycle</i> @e holds tightly on a path <b>iff</b> e occurs at the last state of that path and at no other state of that path.

This represents one cycle of some sampling event. With no explicit sampling event specified, this represents one cycle of the sampling event from the context (i.e., the sampling event from the overall TE or the sampling event for the TCM that contains the TE). When a sampling event is specified, as in *cycle@sampling-event*, this is equivalent to *@sampling-event@sampling-event*.

See also [11.3.1](#) and [11.4](#).

Syntax example:

```
wait cycle
```

### 12.2.3 true(exp)

<b>Purpose</b>	Boolean TE
<b>Category</b>	TE
<b>Syntax</b>	<b>true</b> ( <i>bool</i> : exp)
<b>Parameters</b>	<i>bool</i> A Boolean expression.
<b>Informal semantics</b>	<p>true(<i>exp</i>) @e holds tightly on a given path <b>iff</b></p> <ul style="list-style-type: none"> <li>a)    true(<i>exp</i>) holds at the last state of the path, and</li> <li>b)    cycle @e holds tightly on the path.</li> </ul>

This uses a Boolean expression as a TE. Each occurrence of the sampling event causes an evaluation of the Boolean expression. The Boolean expression is evaluated only at the sampling point.

The TE succeeds each time the expression evaluates to TRUE. The expression *exp* is evaluated after *pclk*. Changes in *exp* after **true**(*exp*) @*pclk* has been evaluated are ignored.

See also [5.1.1](#).

Syntax example:

```
event rst is true(reset == 1) @clk
```

### 12.2.4 @ unary event operator

<b>Purpose</b>	Use an event as a TE
<b>Category</b>	TE
<b>Syntax</b>	@[ <i>struct-exp</i> .] <i>event-type</i>
<b>Parameters</b>	<p><i>struct-exp</i>.                      The name of an event. This can be a predefined event or a user-defined event, and can include the name of the struct instance where the event is defined.</p> <p><i>event-type</i></p>
<b>Informal semantics</b>	<p>@e @q holds tightly on a given path <b>iff</b></p> <ul style="list-style-type: none"> <li>a)    event e occurs at a state of the path, and</li> <li>b)    cycle @q holds tightly on that path.</li> </ul>

An event can be used as the simplest form of a TE. The TE @*event-type* succeeds every time the event occurs. Success of the expression is simultaneous with the occurrence of the event.

The *struct-exp* is an expression that evaluates to the struct instance containing the event instance. If no *struct-exp* is specified, the default is the current struct instance. If a *struct-exp* is included in the event name, the value of the *struct-exp* shall not change throughout the evaluation of the TE.

See also [11.3.1](#) and [11.4](#).

Syntax example:

```
wait @rst
```

### 12.2.5 @ sampling operator (binary @)

<b>Purpose</b>	Specify a sampling event for a TE
<b>Category</b>	TE
<b>Syntax</b>	<i>temporal-expression @event-name</i>
<b>Parameters</b>	<i>temporal-expression</i> A TE.
	<i>event-name</i> The sampling event.
<b>Informal semantics</b>	$t @e$ holds tightly on a given path <b>iff</b> there exist paths $p1$ and $p2$ and state $s$ so that <ol style="list-style-type: none"> <li>the concatenation of <math>p1</math>, <math>s</math>, and <math>p2</math> is the original path;</li> <li><math>t</math> holds tightly on <math>p1</math> <math>s</math>;</li> <li><math>\text{cycle } @e</math> holds tightly on <math>s</math> <math>p2</math>.</li> </ol>

This is used to specify the sampling event for a TE. The specified sampling event overrides the default sampling event. The sampling event applies to all sub-expressions of the TE. It can be overridden for a sub-expression by attaching a different sampling event to the sub-expression. A sampled TE succeeds when its sampling event occurs with or after the success of the TE.

See also [11.3.1](#) and [11.4](#).

Syntax example:

```
wait {@start; cycle} @clk
```

### 12.2.6 and

<b>Purpose</b>	TE and operator
<b>Category</b>	TE
<b>Syntax</b>	<i>temporal-expression and temporal-expression</i>
<b>Parameters</b>	<i>temporal-expression</i> A TE.
<b>Informal semantics</b>	$t1 \text{ and } t2$ holds tightly on a path <b>iff</b> both $t1$ and $t2$ hold tightly on that path.

The temporal **and** succeeds when both TEs start evaluating in the same sampling period and succeed in the same sampling period.

Syntax example:

```
event e12 is (@TE1 and @TE2) @clk
```

### 12.2.7 or

<b>Purpose</b>	TE or operator
<b>Category</b>	TE
<b>Syntax</b>	<i>temporal-expression</i> <b>or</b> <i>temporal-expression</i>
<b>Parameters</b>	<i>temporal-expression</i> A TE.
<b>Informal semantics</b>	$t1$ or $t2$ holds tightly on a path <b>iff</b> either $t1$ or $t2$ holds tightly on that path (or they both hold tightly on that path).

The **or** TE succeeds when either TE succeeds. An **or** operator creates a parallel evaluation for each of its sub-expressions. It can create multiple successes for a single TE evaluation.

Syntax example:

```
event e12 is (@TE1 or @TE2) @clk
```

### 12.2.8 { exp ; exp }

<b>Purpose</b>	TE sequence operator
<b>Category</b>	TE
<b>Syntax</b>	<b>{</b> <i>temporal-expression</i> <b>;</b> <i>temporal-expression</i> <b>;</b> ... <b>}</b>
<b>Parameters</b>	<i>temporal-expression</i> A TE.
<b>Informal semantics</b>	<p><math>\{ t1 ; t2 \}</math> holds tightly on a given path <b>iff</b> there exist paths <math>p1</math> and <math>p2</math> so that</p> <ul style="list-style-type: none"> <li>a) <math>p1</math> concatenated with <math>p2</math> gives the original path;</li> <li>b) <math>t1</math> holds tightly on <math>p1</math>;</li> <li>c) <math>t2</math> holds tightly on <math>p2</math>.</li> </ul>

The semicolon (;) sequence operator evaluates a series of TEs over successive occurrences of a specified sampling event. Each TE following a semicolon (;) starts evaluating in the sampling period following the one where the preceding TE succeeded. The sequence succeeds whenever its final expression succeeds.

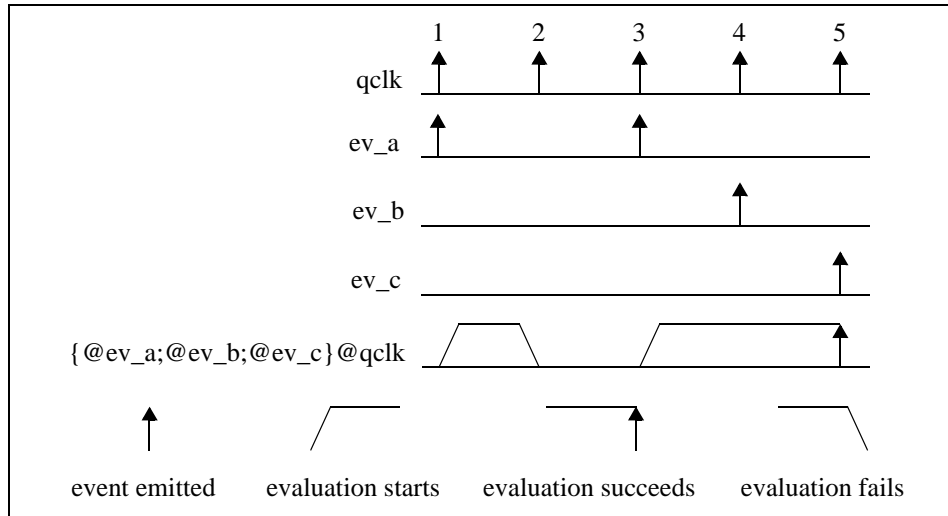
Braces ({ }) are used in the scope of a TE to define a sequence; do not use them in any other way here.

*Example*

[Figure 5](#) shows the results of evaluating the temporal sequence:

```
{@ev_a; @ev_b; @ev_c} @qclk
```

over the series of ev\_a, ev\_b, and ev\_c events shown at the top of [Figure 5](#). The sequence evaluation starts whenever event ev\_a occurs.



**Figure 5—Example evaluations of a temporal sequence**

Syntax example:

```
event de is { @ev_d; @ev_e } @ev_f
```

### 12.2.9 [ exp ]

<b>Purpose</b>	Fixed repetition operator	
<b>Category</b>	TE	
<b>Syntax</b>	<b>[exp]</b> [* <i>temporal-expression</i> ]	
<b>Parameters</b>	<i>exp</i>	A 32-bit, non-negative integer expression that specifies the number of times to repeat the evaluation of the TE. It cannot contain any functions.
	<i>temporal-expression</i>	A TE. If * <i>temporal-expression</i> is omitted, * <code>cycle</code> is automatically used in its place.
<b>Informal semantics</b>	[ <i>exp</i> ] * <i>t</i> is equivalent to { <i>t</i> ; <i>t</i> ; .. ; <i>t</i> }, where the sequence consists of as many terms as the value of <i>exp</i> . A special case is [0] * <i>t</i> , which holds tightly only on the empty path.	

Repetition of a TE is frequently used to describe cyclic or periodic temporal behavior. The [exp] fixed repeat operator specifies a fixed number of occurrences of the same TE. If the numeric expression evaluates to zero (0), the TE succeeds immediately.

Syntax example:

```
wait [2]*cycle
```

### 12.2.10 ~[ exp..exp ]

<b>Purpose</b>	True match variable repeat operator	
<b>Category</b>	Expression	
<b>Syntax</b>	$\sim[[from\text{-}exp]..[to\text{-}exp]] [* \textit{temporal-expression}]$	
<b>Parameters</b>	<i>from-exp</i>	An optional non-negative, 32-bit numeric expression that specifies the minimum number of repetitions of the TE. If the <i>from-exp</i> is missing, zero (0) is used.
	<i>to-exp</i>	An optional non-negative, 32-bit numeric expression that specifies the maximum number of repetitions of the TE. If the <i>to-exp</i> is missing, infinity is used.
	<i>temporal-expression</i>	A TE that is to be repeated some number of times within the <i>from-exp</i> .. <i>to-exp</i> range. If <i>* temporal-expression</i> is omitted, <i>* cycle</i> is automatically used in its place.
<b>Informal semantics</b>	<p>The following definitions apply:</p> <ul style="list-style-type: none"> <li>a) <math>\sim[range]</math> is equivalent to <math>\sim[range]* \textit{cycle}</math></li> <li>b) <math>\sim[ \textit{exp1} .. \textit{exp2} ] * t</math> is equivalent to <math>\{ [ \textit{exp1} ]*t ; \sim[ .. \textit{exp2-exp1} ] * t \}</math></li> <li>c) <math>\sim[ \textit{exp1} .. ] * t</math> is equivalent to <math>\{ [ \textit{exp1} ]*t ; \sim[ .. ] * t \}</math></li> <li>d) <math>\sim[ .. \textit{exp2} ] * t</math> is equivalent to <math>[0]*t \text{ or } [1]*t \text{ or } .. [ \textit{exp2} ]*t</math></li> <li>e) <math>\sim[ .. ] * t</math> is equivalent to <math>[0]*t \text{ or } [1]*t \text{ or } ..</math></li> </ul>	

The true match repeat operator can be used to specify a variable number of consecutive successes of a TE. True match variable repeat succeeds every time the sub-expression succeeds. This expression creates a number of parallel repeat evaluations within the range.

True match repeat also enables specification of behavior over infinite sequences by repeating an infinite number of occurrences of a TE. The expression  $\sim[ .. ] * TE$  is equivalent to:

$[0] \text{ or } [1]*TE \text{ or } [2]*TE ..$

This construct is mainly useful for maintaining information about past events. See also [12.2.9](#).

Syntax example:

```
event p2_4 is ~[2..4]*@pclk
```



12.2.11 [ *exp..exp* ]

<b>Purpose</b>	First match variable repeat operator	
<b>Category</b>	Expression	
<b>Syntax</b>	{ ... ; [[ <i>from-exp</i> .. <i>to-exp</i> ] ] [* <i>repeat-expression</i> ]; <i>match-expression</i> ; ... }	
<b>Parameters</b>	<i>from-exp</i>	An optional non-negative, 32-bit numeric expression that specifies the minimum number of repetitions of the <i>repeat-expression</i> . If the <i>from-exp</i> is missing, zero (0) is used.
	<i>to-exp</i>	An optional non-negative, 32-bit numeric expression that specifies the maximum number of repetitions of the <i>repeat-expression</i> . If the <i>to-exp</i> is missing, infinity is used.
	<i>repeat-expression</i>	A TE that is to be repeated some number of times within the <i>from-exp..to-exp</i> range. If * <i>repeat-expression</i> is omitted, * <i>cycle</i> is automatically used in its place.
	<i>match-expression</i>	The TE to match.
<b>Informal semantics</b>	<p>The following definitions apply:</p> <p>a) { [ <i>range</i> ] * <i>t1</i>; <i>t2</i> } is equivalent to FirstMatch { ~[ <i>range</i> ] * <i>t1</i>; <i>t2</i> }</p> <p>b) FirstMatch <i>t</i> holds tightly on a given path <i>p</i> iff <i>t</i> holds tightly on <i>p</i> has no strict prefix on which <i>t</i> holds tightly.</p> <p>FirstMatch is not part of the <i>e</i> syntax, it is only as an aid for defining the meaning of the first match variable repeat operator.</p>	

The first match repeat operator is only valid in a temporal sequence {TE; TE; TE} (see [12.2.8](#)); it is not a TE operator. The first match repeat expression succeeds on the first success of the *match-expression* between the lower and upper bounds specified for the *repeat-expression*.

First match repeat also enables specification of behavior over infinite sequences by allowing an infinite number of repetitions of the *repeat-expression* to occur before the *match-expression* succeeds. Where @ev\_a is an event occurrence, {[...]\*TE1; @ev\_a} is equivalent to:

{@ev\_a} or {[1]\*TE1; @ev\_a} or {[2]\*TE1; @ev\_a} or {[3]\*TE1; @ev\_a}...

Syntax example:

event ev\_1 is {[2..4]\*@pclk; @reset}

## 12.2.12 fail

<b>Purpose</b>	TE failure operator
<b>Category</b>	TE
<b>Syntax</b>	<b>fail</b> <i>temporal-expression</i>
<b>Parameters</b>	<i>temporal-expression</i> A TE.
<b>Informal semantics</b>	<p><b>fail</b> <i>t</i> holds tightly on a given path <b>iff</b></p> <ul style="list-style-type: none"> <li>a) that path cannot be extended so that <i>t</i> holds tightly on the extended path, and</li> <li>b) the given path does not have a strict prefix on which <b>fail</b> <i>t</i> also holds tightly.</li> </ul>

A **fail** succeeds whenever the TE fails. If the TE has multiple interpretations [e.g., **fail** (TE1 or TE2)], the expression succeeds **iff** all the interpretations fail. The expression **fail** TE succeeds at the point where all possibilities to satisfy TE have been exhausted. Any TE can fail at most once per sampling event.

The **not** operator (see 12.2.16) differs from the **fail** operator, as illustrated in Figure 6.

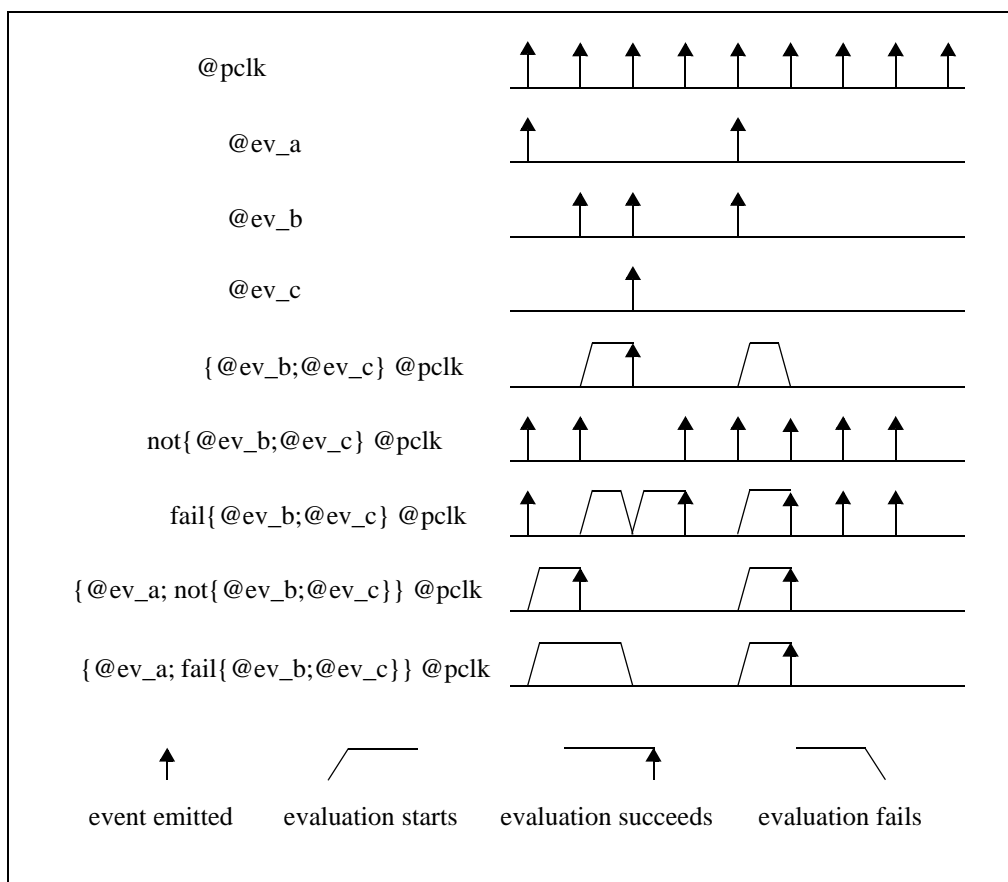


Figure 6—Comparison of temporal not and fail operators

Syntax example:

```
event ev_1 is fail{@ev_b; @ev_c}
```

### 12.2.13 =>

<b>Purpose</b>	Temporal yield operator	
<b>Category</b>	TE	
<b>Syntax</b>	<i>temporal-expression1 =&gt; temporal-expression2</i>	
<b>Parameters</b>	<i>temporal-expression1</i>	The first TE. The second TE is expected to succeed if this expression succeeds.
	<i>temporal-expression2</i>	The second TE. If the first TE succeeds, this expression is also expected to succeed.
<b>Informal semantics</b>	The TE $t1 \Rightarrow t2 @q$ is equivalent to $(fail\ t1\ or\ \{t1;\ t2\}) @q$	

The yield operator is used to assert that success of one TE depends on the success of another TE. The yield expression succeeds without evaluating the second expression if the first expression fails. If the first expression succeeds, then the second expression needs to also succeed in sequence.

The sampling event from the context applies to both sides of the yield operator expression. The entire expression is essentially a single TE, so that  $(TE1 \Rightarrow TE2)@sampling\_event$  is effectively  $(TE)@sampling\_event$ , where TE is the TE made up of  $TE1 \Rightarrow TE2$ .

NOTE—The yield operator is typically used along with the **expect** struct member (see [13.2](#)) to express temporal rules.

Syntax example:

```
expect @A => {[1..2]*@clk; @B}
```

### 12.2.14 eventually

<b>Purpose</b>	TE success check	
<b>Category</b>	TE	
<b>Syntax</b>	<b>eventually</b> <i>temporal-expression</i>	
<b>Parameters</b>	<i>temporal-expression</i>	A TE.
<b>Informal semantics</b>	The TE eventually $t @q$ holds tightly on a given path $p$ <b>iff</b> a) $\{ \sim[ \dots ]; t \} @q$ holds tightly on $p$ , and b) the event <b>quit</b> does not occur at any state of $p$ that precedes the last state of $p$ .	

This is used to indicate the TE is expected to succeed at some unspecified time. Typically, **eventually** is used in an **expect** struct member to specify a TE is expected to succeed sometime before the **quit** event occurs for the struct. See also [28.2.2.5](#).

Syntax example:

```
expect @req => eventually @ack
```

### 12.2.15 detach

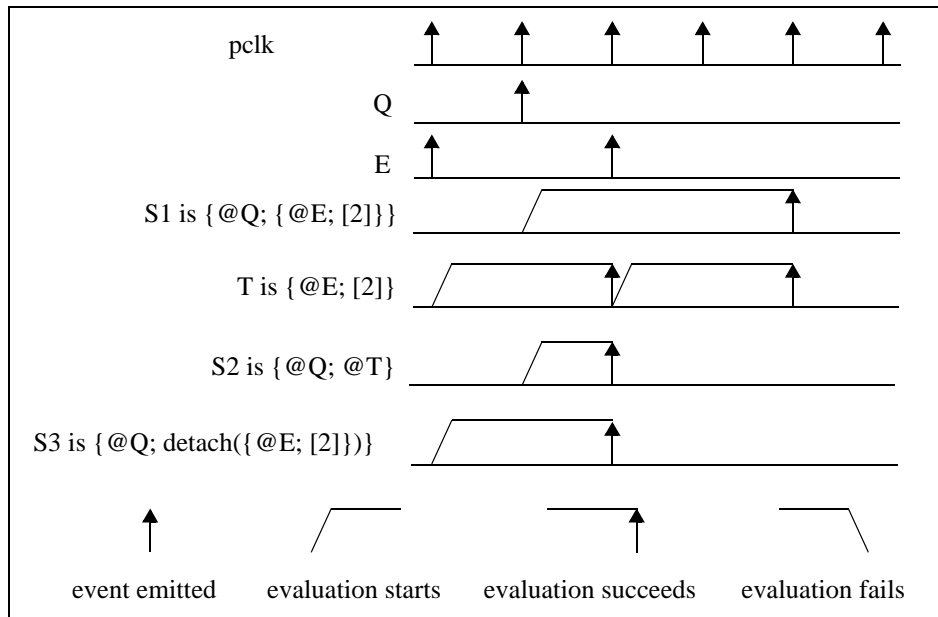
<b>Purpose</b>	Detach a TE
<b>Category</b>	TE
<b>Syntax</b>	<b>detach</b> ( <i>temporal-expression</i> )
<b>Parameters</b>	<i>temporal-expression</i> A TE to be independently evaluated.
<b>Informal semantics</b>	<p>The TE            detach( t1 )  is equivalent to            @unnamed_event_1  where unnamed_event_1 is defined as            event unnamed_event_1 is t1;</p>

A detached TE is evaluated independently of the expression in which it is used. It starts evaluation when the main expression does. Whenever the detached TE succeeds, it emits an “implicit” event that is only recognized by the main TE. The detached TE inherits the sampling event from the main TE.

#### Example

In the following example, both S1 and S2 start with @Q. However, the S1 TE expects E to follow Q, while the S2 TE expects E to precede Q by one cycle. The **detach()** construct causes the TEs to be evaluated separately. As a result, the S3 TE is equivalent to the S2 expression, as shown in [Figure 7](#).

```
struct s {
    event pclk is @sys.pclk;
    event Q;
    event E;
    event T is {@E; [2]} @pclk;
    event S1 is {@Q; {@E; [2]}} @pclk;
    event S2 is {@Q; @T} @pclk;
    event S3 is {@Q; detach({@E; [2]})} @pclk
}
```



**Figure 7—Examples illustrating detached temporal expressions**

Syntax example:

```
@trans.end => detach({@trans.start; ~[2..5]})@pclk
```

#### 12.2.16 not

<b>Purpose</b>	TE inversion operator
<b>Category</b>	TE
<b>Syntax</b>	<b>not</b> <i>temporal-expression</i>
<b>Parameters</b>	<i>temporal-expression</i> A TE.
<b>Informal semantics</b>	The TE not t is equivalent to detach( fail t )

The **not** TE succeeds if the evaluation of the sub-expression does not succeed during the sampling period. Thus, **not** TE succeeds on every occurrence of the sampling event if TE does not succeed.

NOTE—If an event is explicitly *emitted* (see [11.3.2](#)), a race condition can arise between the event occurrence and the **not** of the event when used in some TEs.

Syntax example:

```
event bc_bar is not { @ev_b; @ev_c }
```

12.2.17 **change(exp), fall(exp), rise(exp)**

<b>Purpose</b>	Transition or edge TE
<b>Category</b>	TE
<b>Syntax</b>	<b>change</b>   <b>fall</b>   <b>rise</b> ( <i>scalar</i> : exp) [ <i>@event-type</i> ] <b>change</b>   <b>fall</b>   <b>rise</b> ('HDL-pathname') <b>@sim</b>
<b>Parameters</b>	<i>scalar</i> A Boolean expression or an integer expression.
	<i>event-type</i> The sampling event for the expression.
	<i>HDL-pathname</i> An HDL object enclosed in single quotes ( ' ' ).
	<b>@sim</b> A special annotation used to detect changes in HDL signals.
<b>Informal semantics</b>	<p>a) The following definitions apply for the first syntax type:  <b>change</b>   <b>fall</b>   <b>rise</b>(<i>scalar</i>: exp) [<i>@event-type</i>]</p> <ol style="list-style-type: none"> <li>1) <b>change</b>(exp) @q is equivalent to  true( prev(exp,q) != exp )@q</li> <li>2) <b>fall</b>(exp) @q is equivalent to  true( prev(exp,q) &gt; exp )@q</li> <li>3) <b>rise</b>(exp) @q is equivalent to  true( prev(exp,q) &lt; exp )@q</li> <li>4) <b>prev</b>(exp, q) is a function that gives the value of exp at the previous state for which q holds. If no such state exists, it gives the value of exp at the first state of the path.  <b>prev</b>( ) is not part of the <i>e</i> syntax, it is only an aid for defining the meaning of <b>rise</b>(exp) @q.</li> </ol> <p>b) The following definitions apply for the second syntax type:  <b>change</b>   <b>fall</b>   <b>rise</b>('HDL-pathname') <b>@sim</b></p> <p>— The TE <b>rise</b>('HDL-pathname') <b>@sim</b> serves two main purposes.</p> <ol style="list-style-type: none"> <li>1) It causes the simulator to pass control back to the <i>e</i> program at the end of a simulator delta cycle in which <i>HDL-pathname</i> changes (see <a href="#">11.4.2</a>).</li> <li>2) It is a TE.</li> </ol> <p>If <b>rise</b>('HDL-pathname') <b>@sim</b> appears as a top-level TE, it is already in sampled normal form.</p> <p>— <b>rise</b>('HDL-pathname') <b>@sim</b> holds tightly on a path <b>iff</b></p> <ol style="list-style-type: none"> <li>1) the last point of the path corresponds to a tick in which the signal <i>HDL-pathname</i> changes;</li> <li>2) the signal changes at no other point of the path;</li> <li>3) the signal's new value is larger than its old value.</li> </ol> <p>— <b>fall</b> and <b>change</b> perform in a similar fashion.</p>

This detects a change in the sampled value of an expression. The expression is evaluated at each occurrence of the sampling event and compared to the value it had at the previous sampling point. Only the values at sampling points are detected. The value of the expression between sampling points is invisible to the TE.

[Table 29](#) describes the behavior of each of the three TEs (**change**, **fall**, and **rise**). When applied to HDL variables, the expressions examine the value after each bit is translated from the HDL four-value or nine-value logic representation to *e* two-value logic representation.

**Table 29—Edge condition options**

Edge condition	Meaning
<b>rise</b> ( <i>exp</i> )	Triggered when the expression changes from FALSE to TRUE. If it is an integer expression, the <b>rise</b> () TE succeeds upon any change from $x$ to $y > x$ . Signals wider than one bit are allowed. Integers larger than 32 bits are not allowed.
<b>fall</b> ( <i>exp</i> )	Triggered when the expression changes from TRUE to FALSE. If it is an integer expression, the <b>fall</b> () TE succeeds upon any change from $x$ to $y < x$ . Signals wider than one bit are allowed. Integers larger than 32 bits are not allowed.
<b>change</b> ( <i>exp</i> )	Triggered when the value of the expression changes. The <b>change</b> () TE succeeds upon any change of the expression. Signals wider than one bit are allowed. Integers larger than 32 bits are not allowed.

[Table 29](#) describes the default translation of HDL nine-value logic representation to *e* values. The @x and @z HDL value modifiers can be used to override the default translation (see [24.3](#)).

**Table 30—Transition of HDL values**

HDL values	<i>e</i> value
0, X, U, W, L, -	0
1, Z, H	1

A special notation, @sim, can be used in place of a sampling event for the **rise**, **fall**, or **change** of HDL objects. If @sim is used, the HDL object is watched by the simulator. The @sim notation does not signify an event, but is used only to cause a callback any time there is a change in the value of the HDL object to which it is attached.

When a sampling event other than @sim is used, changes to the HDL object are detected only if they are visible at the sampling rate of the sampling event (see [Figure 8](#)).

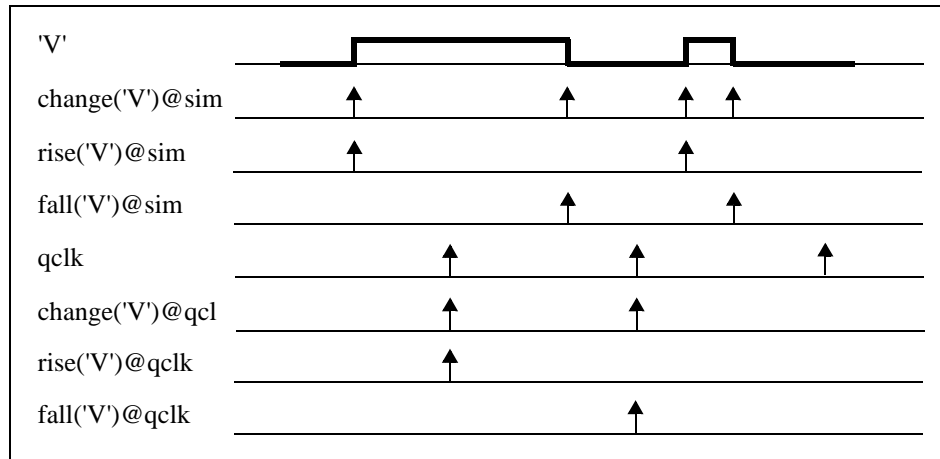
NOTE—An *e* program ignores glitches (see [11.4.2](#)) that occur in a single simulation time slot. Only the first occurrence of a particular monitored event in a single simulation time slot is recognized by the *e* program.

Syntax example:

```
event hv_c is change('top.hold_var')@sim
```

*Example*

[Figure 8](#) shows evaluations of the **rise**, **fall**, and **change** expressions for the HDL signal *V*, with the sampling events @sim and @qclk. The qclk event is an arbitrary event that is emitted at the indicated points. The *V* signal rises and then falls between the second and third occurrences of event qclk. Since the signal's value is the same at the third qclk event as it was at the second qclk event, the **change**('V')@qclk expression does not succeed at the third qclk event.



**Figure 8—Effects of the sampling rate on detecting HDL object changes**

### 12.2.18 delay

<b>Purpose</b>	Specify a simulation time delay	
<b>Category</b>	TE	
<b>Syntax</b>	<b>delay</b> ( <i>int</i> : <i>exp</i> )	
<b>Parameters</b>	<i>int</i>	An integer expression or time expression no larger than 64 bits. The number specifies the amount of simulation time to delay. The time units are in the timescale used in the HDL simulator.
<b>Informal semantics</b>	delay( <i>exp</i> ) holds tightly on a path whose elapsed simulation time is <i>exp</i> .	

This succeeds after a specified simulation time delay elapses. A callback occurs after the specified time. A delay of zero (0) succeeds immediately. See also [14.1.3](#).

Attaching a sampling event to **delay** has no effect. The **delay** ignores the sampling event and succeeds as soon as the delay period elapses. Also, this expression can only be used if the *e* program is being run with an attached HDL simulator.

Syntax example:

```
wait delay(3)
```



### 12.2.19 exec

<b>Purpose</b>	Attach an action block to a TE	
<b>Category</b>	TE side effect	
<b>Syntax</b>	<i>temporal-expression</i> <b>exec</b> <i>action</i> ; ...	
<b>Parameters</b>	<i>temporal-expression</i>	The TE that invokes the action block.
	<i>action</i> ; ...	A series of actions to perform when the expression succeeds.

This invokes an action block when a TE succeeds. The action block cannot contain any time-consuming actions. The actions are executed immediately upon the success of the expression, but not more than once per tick. To support extensibility, use method calls in the **exec** action block rather than calling the actions directly.

The usage of **exec** is similar to the **on** struct member, except:

- any TE can be used as the condition for **exec**, while only an event can be used as the condition for **on**;
- **exec** needs to be used in the context of a TE in a TCM or an event definition, while **on** can only be a struct member.

An **exec** action cannot be attached to a first match variable repeat expression. However, it can be attached to the repeat expression of a first match variable repeat expression. An **exec** action also cannot be attached to an implicit repeat expression, it needs to be attached to an explicit repeat expression.

See also [4.2.3](#) and [13.1](#).

Syntax example:

```
wait @watcher_on exec {print watcher_status_1}
```

## 12.3 Success and failure of a temporal expression

The meaning of TEs is defined in terms of tight satisfaction (“holds tightly”), as seen in [12.2](#). For top-level TEs, it is useful to introduce the notions of “success” and “failure,” which are derived from tight satisfaction. The notion of *success* is used for describing the meaning of **event** struct members and **wait** and **sync** actions. The notion of *failure* is used to describe the meaning of **assume** and **expect** struct members.

### 12.3.1 Success of a temporal expression

A TE *succeeds* at a point of a path if it holds tightly on a sub-path ending in the given point. More precisely:

A TE  $\tau$ , whose sampling event is  $q$ , succeeds at the  $i^{\text{th}}$  state of a path  $p = s_0 s_1 \dots s_i \dots s_n$ ,  $0 \leq i \leq n$ , **iff**  
 either  $\tau$  holds tightly on  $s_0 \dots s_i$  or else there is a  $j$ ,  $0 < j \leq i$  such that event  $q$  holds at  $s_{j-1}$  and  $\tau$  holds tightly on  $s_j \dots s_i$ .

The notion of success of a TE can also be used without specifying a path and a state. Then it is presumed the path corresponds to the execution of a given *e* program and the state corresponds to the current state of the execution.

### 12.3.2 Failure of a temporal expression

A TE  $\tau$ , whose sampling event is  $q$ , *fails* at the  $i^{\text{th}}$  state of a path **iff** `fail`  $\tau$  succeeds at the  $i^{\text{th}}$  state of that path. Again, if no path and state are specified, it is presumed the path corresponds to the execution of a given  $e$  program and the state corresponds to the current state of the execution.

### 12.3.3 Start of an evaluation of a temporal expression

For a given path  $p = s_0 s_1 \dots s_i \dots s_n$ , a state  $s_i$   $0 \leq i \leq n$  of that path, and a TE  $\tau$ , whose sampling event is  $q$ , the definition of success of  $\tau$  on  $p$  at  $s_i$  considers various sub-paths of  $p$  and tests for *tight satisfaction* of  $\tau$  on these sub-paths. The sub-paths considered are those sub-paths that start with a state just after a state in which  $q$  occurs and end in  $s_i$ , and also the sub-path that starts with the first state of  $p$  and ends in  $s_i$ .

The evaluation of  $\tau$  starts at the first state of  $p$  and also at any states following a state where the sampling event occurs. The TE can succeed only in those states in which the sampling event occurs, except for TEs that hold tightly on the empty path, e.g., `[0]*cycle @q`.

The paths over which a top-level TE is interpreted during the execution of an  $e$  program depend on the construct in which the TE appears.

- For **event**, **expect**, and **assume** struct members, evaluation of the TE starts as soon as the parent struct is generated and ends when the parent struct is quit.
- For **sync** actions, the TE is evaluated as soon as the **sync** action is reached.
- For **wait** actions, the evaluation of the TE starts in the tick after reaching the **wait** action.
- For both **sync** and **wait** actions, evaluation ends if the TE succeeds or the parent TCM is terminated.

## 13. Temporal struct members

In addition to the **event** struct member (see [11.3.1](#)), there are two other struct members used for temporal coding: **on** and **expect** | **assume**.

### 13.1 on

<b>Purpose</b>	Specify a block of actions that execute on an event	
<b>Category</b>	Struct member	
<b>Syntax</b>	<b>on</b> <i>event-type</i> { <i>action</i> ; ...}	
<b>Parameters</b>	<i>event-type</i>	The name of an event that invokes the action block.
	<i>action</i> ; ...	A block of non-time-consuming actions.

This defines a struct member that executes a block of actions immediately whenever a specified event occurs. An **on** struct member is similar to a regular method, except the action block for an **on** struct member is executed before TCMs waiting for the same event. The actions are executed in the order in which they appear in the action block.

To extend an **on** struct member, repeat its declaration and use a different action block. This has the same effect as using **is also** to extend a method.

The **on** struct member is implemented as a method, named **on\_event-type()**. To invoke the action block without the occurrence of the event, call the **on\_event-type()** method. This method can be extended like any other method, by using **is**, **is also**, **is only**, or **is first** (see [18.1.3](#)).

The following restrictions also apply:

- The named event shall be local to the struct in which the **on** is defined.
- The **on** action block shall not contain any time-consuming actions or TCMs.

See also [4.2.3](#) and [18.1.3](#).

Syntax example:

```
on xmit_ready {
    transmit()
}
```

### 13.2 **expect** | **assume**

<b>Purpose</b>	Define a temporal rule	
<b>Category</b>	Struct member	
<b>Syntax</b>	<b>expect</b>   <b>assume</b> [ <i>rule-name</i> <b>is</b> [ <b>only</b> ]] <i>temporal-expression</i> [ <b>else dut_error</b> ( <i>string-exp</i> )] <b>expect</b>   <b>assume</b> <i>rule-name</i>	
<b>Parameters</b>	<i>rule-name</i>	A name that uniquely identifies the rule from other rules or events within the struct. It can be used to override the temporal rule later on in the code or change from <b>expect</b> to <b>assume</b> or vice versa.
	<i>temporal-expression</i>	A TE that is always expected to succeed. Typically involves a temporal yield ( $\Rightarrow$ ) operator (see <a href="#">12.2.13</a> ).
	<i>string-exp</i>	A string or a method that returns a string. If the TE fails, the string is printed, or the method is executed and its result is printed.

Both the **expect** and **assume** struct members are used for defining temporal properties.

- When a simulation-based tool executes the *e* program, it evaluates the rule expressed by the TE. If the TE fails at some point in time (see [12.3](#)), the tool reports an error as specified with the **dut\_error** clause (if no **dut\_error** clause is specified, the tool prints the rule name). The notion of failure of the TE implies a new evaluation starts on every state following a state in which the sampling event occurs (see [12.3.3](#)). Simulation-based tools typically treat **expect** and **assume** in exactly the same manner.
- When a formal verification tool analyzes the *e* program, **expect** struct members are interpreted as rules the tool needs to verify, whereas **assume** struct members are interpreted as constraints on legal behavior. This means the tool looks for program execution paths where the TE bound to an **expect** fails (see [12.3](#)) and none of the temporal expressions (TEs) bound to the **assumes** fail.

Once a rule has been defined, it can be modified using the **is only** syntax and can be changed from an **expect** to an **assume** or vice versa. To perform multiple verification runs, vary the rules slightly or use the same set of rules in different **expect/assume** combinations.

Syntax example:

```
expect @a => {[1..5]; @b} @clk
```

*Example*

This example defines an **expect**, `bus_cycle_length`, that requires the length of the bus cycle to be no longer than 1000 cycles.

```
struct bus_e {
    event bus_clk          is change('top.b_clk') @sim;
    event transmit_start is rise ('top.trans') @bus_clk;
    event transmit_end   is rise ('top.transmit_done') @bus_clk;
    event bus_cycle_length;

    expect bus_cycle_length is
        @transmit_start => {[0..999]; @transmit_end} @bus_clk
    else dut_error("Bus cycle did not end in 1000 cycles")
}
```

## 14. Time-consuming actions

*Time-consuming actions* are distinguished actions that shall only appear in the body of a TCM (see [Clause 18](#)); however, they shall not appear in any of the following contexts, even if they are themselves embedded within a TCM:

- **exec** action block (see [12.2.19](#))
- **new .. with** action block (see [4.16.2](#))
- **error()** action block (see [17.3.2](#))

Any attempt to use a time-consuming action in an illegal context shall cause the compiler to issue an error message.

### 14.1 Synchronization actions

The **sync** (see [14.1.2](#)) and **wait** (see [14.1.3](#)) actions are used to synchronize temporal test activities within an *e* program, and between the DUT and the *e* program. Both actions operate on an optional TE; if it is omitted, then the TE cycle is implied. See also [Clause 12](#).

NOTE—The **put()** and **get()** operations on buffer ports have blocking behavior and may be used for synchronization as well. See [9.8](#).

#### 14.1.1 Synchronization semantics

The declaration of a TCM introduces a default sampling event (see [Clause 18](#)), which serves as the sampling event for any TE embedded in a **wait** or **sync** action that does not have its own sampling event specified. Thus the following examples are equivalent:

```

tcm() @p_clk is { wait };
tcm() @p_clk is { wait cycle@p_clk }

```

A number of concurrent TCMs can be invoked using the **start** action (see [18.2.2](#)). An *e* program consisting of a number of concurrently executing TCMs simulates concurrency by interleaving their actions over time. The TEs embedded in the synchronization actions orchestrate this behavior as follows:

- a) When an executing TCM reaches a synchronization action, its execution shall be suspended and some other suspended TCM can then be scheduled for execution.
- b) When a suspended TCM is scheduled for execution, the system first evaluates the embedded TE:
  - 1) If this succeeds, the TCM resumes execution immediately;
  - 2) If it remains undecided, the TCM remains suspended;
  - 3) If it fails, the TCM remains idle until the next occurrence of the sampling event.

When a TCM resumes, it commences executing from the first action following the synchronization action and continues execution without interruption until it either terminates normally (for a started TCM) or encounters a subsequent synchronization action.

In this way, the synchronization actions in the body of a TCM delimit the program fragments that are executed atomically without interruption; the system can pause and choose another TCM for execution only when a synchronization action is encountered.

The following restrictions also apply:

- If a called TCM returns, the callee resumes execution immediately.
- When a TCM is called or started, there is an implicit “sync cycle” before it commences execution.

### 14.1.2 sync

<b>Purpose</b>	Synchronize a TCM to the success of the embedded TE.
<b>Category</b>	Action
<b>Syntax</b>	<b>sync</b> [ <i>temporal-expression</i> ]
<b>Parameters</b>	<i>temporal-expression</i> A TE that specifies how the synchronization is achieved.

When an TCM reaches a **sync** action, its execution shall be suspended so some other TCM, or the suspended TCM itself, can be scheduled for execution.

If the *temporal-expression* is omitted, *cycle* is assumed and the TCM therefore synchronizes to its default sampling event.

Syntax example:

```
sent_data_tcm();
sync
```

### 14.1.3 wait

<b>Purpose</b>	Wait until a TE succeeds
<b>Category</b>	Action
<b>Syntax</b>	<b>wait</b> [[ <b>until</b> ] <i>temporal-expression</i> ]
<b>Parameters</b>	<i>temporal-expression</i> A TE that specifies how the synchronization is achieved.

When a TCM reaches a **wait** action, its execution shall be suspended so some other TCM can be scheduled for execution. The suspended TCM itself shall remain idle and shall not be scheduled until the next occurrence of the sampling event.

If the *temporal-expression* is omitted, *cycle* is assumed and the TCM therefore waits for the next occurrence of the default sampling event.

NOTE—The **until** option is an aid to readability only and has no effect on the semantics of the action.

Syntax example:

```
wait [3]*cycle
```

## 14.2 Concurrency actions

The **all of** (see [14.2.1](#)) and **first of** (see [14.2.2](#)) actions facilitate concurrent execution within TCMs. They introduce explicit concurrent branches whose actions shall be interleaved in the same way that the actions of concurrent TCMs are themselves interleaved. The parallel action blocks that form the body of the **all of** and **first of** actions can start or call TCMs, synchronize through **wait** and **sync** actions, and execute all normal and time-consuming actions subject to the following limitations:

- It is illegal to execute a **return** action (see [18.2.5](#)) from the body of a concurrency action.
- It is illegal to execute either a **break** or **continue** action (see [21.4](#)) from the body of a concurrency action, unless the loop itself is embedded in the body of the concurrency action.

The compiler shall issue an error message in each of the preceding circumstances.

#### 14.2.1 all of

<b>Purpose</b>	Execute action blocks in parallel
<b>Category</b>	Action
<b>Syntax</b>	<b>all of</b> <code>{{action; ...}; ... }</code>
<b>Parameters</b>	<code>{action; ...};</code> Action blocks that are to execute concurrently. Each action block is a separate branch.

This executes multiple action blocks concurrently as separate branches of a fork, continuing with subsequent actions only when all the action blocks have completed executing.

When execution of a TCM reaches an **all of** action, the branches shall be executed in some indeterminate order until each branch either terminates normally or reaches a synchronization action (see [14.1](#)) and is suspended. The rules for scheduling and resuming the execution of the branches of an **all of** action are the same as those for concurrent TCMs (see [14.1.1](#)).

The subsequent actions following the **all of** action shall be reached when and only when execution of all the individual branches in the body has completed. When the last branch of the **all of** action completes, the subsequent actions commence execution immediately.

Syntax example:

```
all of {
    { check_bus_controller()    };
    { check_memory_controller() };
    { wait cycle; check_alu()   }
}
```

#### 14.2.2 first of

<b>Purpose</b>	Execute action blocks in parallel
<b>Category</b>	Action
<b>Syntax</b>	<b>first of</b> <code>{{action; ...}; ... }</code>
<b>Parameters</b>	<code>{action; ...};</code> Action blocks that are to execute concurrently. Each action block is a separate branch.

This executes multiple action blocks concurrently as separate branches of a fork, continuing with subsequent actions once one of the action blocks has completed executing.

When execution of a TCM reaches a **first of** action, the branches shall be executed in some indeterminate order until some branch terminates normally or each branch reaches a synchronization action (see [14.1](#)) and

is suspended. The rules for scheduling and resuming the execution of the branches of a **first of** action are the same as those for concurrent TCMs (see [14.1.1](#)).

The subsequent actions following the **first of** action shall be reached when and only when execution of one of the individual branches in the body has completed. When one of the branches of the **first of** action completes, the subsequent actions commence execution immediately and the suspended parallel branches of the **first of** action are terminated (or preempted) immediately.

Syntax example:

```
first of {
  { wait [3]*cycle@ev_a };
  { wait @ev_e }
}
```

## 14.3 State machines

This subclause describes the syntax and semantics of the state machine time-consuming action. The state machine action is used to in-line a finite state machine in the body of a TCM. See also [Clause 18](#).

### 14.3.1 Overview

The **state machine** action is a construct for modeling finite state machines. A *state machine definition* consists of the state machine action block, along with the identification of a state variable that holds the current state of the machine. The *state variable* is a local variable (see [Clause 4](#)) of the enclosing TCM or a field path expression (see [4.3.4](#)); in either case, the entity shall be a scalar of some enumerated type (see [Clause 4](#)). The symbolic values of the enumerated type are the states of the state machine being defined. The rules governing state variables are detailed in [14.3.2](#).

The **state machine** action block specifies the behavior of the state machine. This *action block* consists of a number of special state-transition actions and state actions. A *state-transition action* is a time-consuming action that governs how the state machine changes from one state to another. There are two kinds of state-transition action, as detailed in [14.3.3](#) and [14.3.4](#). A **state action** is a block that is executed whenever the identified state of the machine is entered (see [14.3.5](#)).

In addition, the **state machine** action can nominate a final state of the state machine. If it does so, when the machine enters this state, the **state machine** action shall terminate and subsequent actions shall commence execution.



### 14.3.2 state machine action

<b>Purpose</b>	Define a state machine
<b>Category</b>	Action
<b>Syntax</b>	<b>state machine</b> <i>state-variable</i> [ <b>until</b> <i>final-state</i> ] {( <i>state-transition-action</i>   <i>state-action</i> ); ...}
<b>Parameters</b>	<i>state-variable</i> This is a local variable (or a field) of some enumerated type.
	<i>final-state</i> The state at which the state machine normally terminates.
	<i>state-transition-action</i> A state transition that occurs when the associated action block finishes. See <a href="#">14.3.3</a> and <a href="#">14.3.4</a> .
	<i>state-action</i> Actions executed upon entering the named state (see <a href="#">14.3.5</a> ).

This action declares a state machine using a *state-variable* of some enumerated type to hold the current state of the machine. The states of the machine are the symbolic values of this variable's enumerated type. When the **state machine** action is reached, the state machine starts in the first state listed in the state variable's enumerated type. If the optional **until** *final-state* exit condition is used, the state machine runs until that state is reached, whereupon it terminates. *final-state* shall be one of the values of the state-variable's enumerated type. If the **until** clause is not used, the state machine runs until the enclosing construct, usually a TCM action block or a concurrency action, is terminated in some way.

The definition of the *state machine* consists of a number of state-transition actions and state actions, as follows:

- a) Simple state transition action:      *state* => *state* {*action*; ...}
- b) Wild card state transition action:    *\*=>* *state* {*action*; ...}
- c) State action:                          *state* {*action*; ...}

In each case, the state shall be one of the legal enumerated values of the *state-variable* used in the definition of the state machine. The action block associated with the *state* or *state-transition* actions can contain any legal actions, time-consuming or otherwise, subject to the following limitations:

- It is illegal to use a **return** action (see [18.2.5](#)) in a *state* or *state-transition* action block.
- It is illegal to use a **break** or **continue** action (see [21.4](#)) in a *state* or *state-transition* action block, unless enclosed by a **loop** action within the block.

The compiler shall issue an error message in each of the preceding circumstances.

The effect of assigning to the *state-variable* used in the definition of the state machine in any of the *state* or *state-transition* action blocks is undefined. Nested state machines shall use distinct *state* variables to hold their state; otherwise, the result is undefined.

Syntax example:

```
var stv : [initial, running, done];

state machine stv until done {
    initial => running {
        wait until rise('top.a')
    };
};
```

```

initial => done {
    wait until change('top.r1');
    wait until rise('top.r2')
};

running => initial {
    wait until rise('top.b')
};

running {
    out("Entered ", stv, " state at ", sys.time);
    while TRUE do {
        wait cycle;
        out("still running")
    }
}

```

### 14.3.3 Simple state transition: state => state

<b>Purpose</b>	One-to-one state transition	
<b>Category</b>	State transition	
<b>Syntax</b>	<i>current-state</i> => <i>next-state</i> { <i>action</i> ; ...}	
<b>Parameters</b>	<i>current-state</i>	The state from which the transition starts.
	<i>next-state</i>	The state to which the transition changes.
	<i>action</i> ; ...	The sequence of actions that precede the transition.

This specifies how a transition occurs from one state to another. The action block starts executing when the state machine enters the current state. When the action block completes, the transition to the next state occurs. The action block usually contains at least one time-consuming action; if it does not, the transition to the next state is immediate.

If the *current-state* is applicable to two or more simple or wild card *state-transition* actions, then each action block commences execution concurrently when the *current-state* is entered; in this case, the actual next state chosen is indeterminate and depends upon the first of the action blocks to complete execution. When the state transition occurs, any concurrent action blocks associated with the *current-state* shall be preempted.

Syntax example:

```

begin => run {
    wait [2]*cycle;
    out("Run state entered")
}

```

#### 14.3.4 Wild card state transition: $\ast \Rightarrow$ state

<b>Purpose</b>	Any-to-one state transition	
<b>Category</b>	State transition	
<b>Syntax</b>	$\ast \Rightarrow$ <i>next-state</i> { <i>action</i> ; ...}	
<b>Parameters</b>	<i>next-state</i>	The state to which the transition changes.
	<i>action</i> ; ...	The sequence of actions that precede the transition.

This specifies how a transition occurs from any defined state to a particular state. The action block starts executing when the state machine enters a new state. When the action block completes, the transition to the next state occurs.

If two or more simple or wild card *state-transition* actions apply to any state of the machine, then each action block commences execution concurrently when that state is entered; in this case, the actual next state chosen is indeterminate and depends upon the first of the action blocks to complete execution. When the state transition occurs, any concurrent action blocks associated with that state shall be preempted.

Syntax example:

```
* => pause {
    wait @sys.restart;
    out("Entering pause state")
}
```

#### 14.3.5 state action

<b>Purpose</b>	Execute actions upon entering a state, with no state transition	
<b>Category</b>	State action block	
<b>Syntax</b>	<i>current-state</i> { <i>action</i> ; ...}	
<b>Parameters</b>	<i>current-state</i>	The state for which the action block is to be executed.
	<i>action</i> ; ...	The sequence of actions that is executed upon entering the current state.

This specifies an action block that is executed when a specific state is entered. No transition occurs when the action block completes.

- If the *current-state* is also the final state of the state machine, then the state machine terminates when the action block completes, and any actions subsequent to the state machine action shall then commence execution immediately.
- There can be more than one state action associated with the *current-state*; in this case, all the action blocks commence execution concurrently when the *current-state* is entered.
- If the *current-state* has one or more *state-transition* actions associated with it, then the state action block and the relevant *state-transition* actions all commence execution concurrently when the *current-state* is entered. If some *state-transition* action block completes execution, the state transition occurs immediately, possibly preempting the state action block if it has not already completed execution.

Syntax example:

```
* => run {
    out("* to run");
    wait cycle
};

run {
    out("In run state");
    wait cycle;
    out("Still in run")
};

run => done {
    out("run to done");
    wait cycle
}
```

## 15. Coverage constructs

This clause describes how to define, extend, and use coverage constructs. See also [6.2](#).

### 15.1 Defining coverage groups: cover

<b>Purpose</b>	Define a coverage group	
<b>Category</b>	Struct member	
<b>Syntax</b>	<b>cover</b> <i>event-type</i> [ <b>using</b> <i>coverage-group-option</i> , ...] <b>is</b> { <i>coverage-item-definition</i> ; ...} <b>cover</b> <i>event_type</i> <b>is empty</b>	
<b>Parameters</b>	<i>event-type</i>	The name of the group. This shall be the name of an event type defined previously in the struct. The event shall not have been defined in a subtype. However, if the event and coverage group are defined in the same file, the event definition does not have to appear before the coverage group definition. The event is the sampling event for the coverage group. Coverage data for the group is collected every time the event is emitted. The full name of the coverage group is <i>struct-exp.event-name</i> . The full name shall be specified for the coverage methods.
	<i>coverage-group-option</i>	The coverage group options listed in <a href="#">Table 31</a> can be set via the <b>using</b> keyword. Each coverage group can have its own set of options. The options can appear in any order after the <b>using</b> keyword.
	<i>coverage-item-definition</i>	The definition of a coverage item (see <a href="#">15.2</a> ).

This defines a coverage group. A *coverage group* is a struct member that contains a list of data items for which data is collected over time. Once data coverage items have been defined in a coverage group, they can be used to define special coverage group items called **transition** and **cross** items (see [15.4](#) and [15.3](#), respectively). The **is** keyword can also be used to define a new coverage group (see [15.5](#) for information on using **is also** to extend an existing coverage group).

The sampling event of a coverage group cannot be defined in a **when** subtype of a struct. However, a coverage group that uses an event defined in a parent struct can be defined in a **when** subtype. If they are extended by adding a **per\_instance** item, the instances refer only to the **when** subtype. If a **per\_instance** item is instead defined in a base type and additional items are added under the **when** construct, then the cover group instances refer to the base type and the cover item values refer to the **when** subtype (see [15.2.1](#)).

The **empty** keyword can be used to define an empty coverage group that will be extended later by using a **cover is also** struct member with the same name (see [15.5](#)). See also [Clause 11](#) and [Clause 6](#).

**Table 31—Coverage group options**

Option	Description
<b>no_collect</b>	This coverage group is not displayed in coverage reports and is not saved in the coverage files.

**Table 31—Coverage group options (*continued*)**

Option	Description
<b>count_only</b>	This option reduces memory consumption because the data collected for this coverage group is reduced. Interactive, post-processing cross coverage of items is not allowed in this case. The coverage configuration option <b>count_only</b> (see <a href="#">29.9</a> ) sets this option for all coverage groups.
<b>text=string</b>	A text description for this coverage group. This can only be a quoted string (" "), not a variable or expression. The text is shown at the beginning of the information for the group in the coverage report.
<b>when=bool-exp</b>	The coverage group is sampled only when <i>bool-exp</i> is TRUE. The <i>bool-exp</i> is evaluated in the context of the parent struct, i.e., the scope of entities is taken from that context. Concretely, names of fields shall be taken as fields of the parent structs, and methods called shall be executed in the context of said struct. The pseudo-variable <b>me</b> refers to that struct.
<b>global</b>	A <i>global coverage group</i> is a group whose sampling event is expected to be emitted only once. If more than one sampling event is emitted, a DUT error shall be issued. If items from a global group are used in interactive cross coverage, no timing relationships exist between the items.
<b>radix=DEC   HEX   BIN</b>	Buckets for items of type <b>int</b> or <b>uint</b> are given the item value ranges as names. This option specifies in which radix the bucket names are displayed. The global <b>print</b> radix option does not affect the bucket name radix. Legal values are <b>DEC</b> (decimal), <b>HEX</b> (hexadecimal), and <b>BIN</b> (binary). The value shall be in uppercase letters. If the <b>radix</b> is not used, <b>int</b> or <b>uint</b> bucket names are displayed in decimal.
<b>weight=uint</b>	This option specifies the grading weight of the current group relative to other groups. It is a non-negative integer with a default of 1.

NOTE—Unless coverage mode is turned on first (see [29.9](#)), no coverage results are collected, even if cover groups and cover items are defined.

Syntax example:

```
cover inst_driven is {
    item opcode;
    item op1;
    cross opcode, op1
}
```

## 15.2 Defining basic coverage items: item

<b>Purpose</b>	Define a coverage item
<b>Category</b>	Coverage group item
<b>Syntax</b>	<b>item</b> <i>item-name</i> [: <i>type=exp</i> ] [ <b>using</b> <i>coverage-item-option</i> , ...]
<b>Parameters</b>	<i>item-name</i> The name assigned to the coverage item. If the optional <i>type=exp</i> is not used, the value of the field named <i>item-name</i> is used as the coverage sample value. The field can be a scalar, not larger than 32 bits, or a string. If <i>type=exp</i> is specified, the value of <i>exp</i> is used as the coverage sample value.
	<i>type</i> The type of the item. The type expression shall evaluate to a scalar not larger than 32 bits or to a string.
	<i>exp</i> The expression is evaluated at the time the whole coverage group is sampled. If the <b>using when</b> option is used, expression evaluation is further restricted by the <b>when</b> expression evaluating to TRUE. The value of <i>exp</i> is recorded for the item.
	<i>coverage-item-option</i> The coverage group options listed in Table 32 can be set via the <b>using</b> keyword. The options can appear in any order after the <b>using</b> keyword.

**Table 32—Coverage item options**

Option	Description
<b>per_instance</b>	Coverage data is collected and graded for all the other items in a separate listing for each bucket of this item. See also <a href="#">15.2.1</a> and <a href="#">15.2.2</a> .
<b>no_collect</b>	This coverage item is not displayed in coverage reports and is not saved in the coverage files. However, it can be referenced by cross and transition items.
<b>text=string</b>	A text description for this coverage item. This can only be a quoted string (" "), not a variable or expression. In an ASCII coverage report, the text is shown along with the item name.
<b>when=bool-exp</b>	The item is sampled only when <i>bool-exp</i> is TRUE. The <i>bool-exp</i> is evaluated in the context of the parent struct, i.e., the scope of entities is taken from that context. Concretely, names of fields shall be taken as fields of the parent structs and methods called shall be executed in the context of said struct. The pseudo-variable <b>me</b> refers to that struct. The sampling is done at runtime.
<b>at_least=num</b>	The minimum number of samples for each bucket of the item. Anything less than <i>num</i> is considered a hole. This option cannot be used for an <i>ungradeable item</i> , an item whose number of buckets exceeds the configuration option <b>max_int_buckets</b> . This shall be a non-negative number; the default is 1.

Table 32—Coverage item options (*continued*)

Option	Description
<b>ranges</b> = <b>{range(parameters)</b> <b>[: range(parameters) ...]}</b>	<p>This option creates buckets for this item's values or ranges of values. It cannot be used for string items.</p> <p><b>range()</b> can have one, two, three, or four parameters that specify how the values are separated into buckets. The first parameter, <i>range</i>, is required. The other three are optional. The syntax for range options is:</p> <p style="text-align: center;"><b>range(range: range, [name: string, every-count: int, at-least-num: int])</b></p> <p>The parameters are:</p> <ul style="list-style-type: none"> <li>— <i>range</i> The range for the bucket. It shall be a literal range, such as [ 1 . . 5 ], of the proper type. Even a single value needs to be specified in brackets, e.g., [ 7 ]. If overlapping ranges are specified, the values of the overlapping region go into the first of the overlapping buckets. The specified range for a bucket is the bucket name.</li> <li>— <i>name</i> A name for the bucket. If this parameter is used, the <i>every-count</i> parameter shall be set to UNDEF.</li> <li>— <i>every-count</i> The size of the buckets to create within the range. If this parameter is used, the <i>name</i> parameter shall be set to an empty string ( " " ).</li> <li>— <i>at-least-num</i> A number that specifies the minimum number of samples required for a bucket. If the item occurs fewer times than this, a hole is marked. This parameter overrides the global <b>at_least</b> option and the per-item <b>at_least</b> option. The value of <i>at-least-num</i> can be set to zero (0), meaning "do not show holes for this range."</li> </ul>
<b>ignore</b> =item-bool-exp	<p>Defines values that are to be completely ignored. They do not appear in the statistics at all. The expression is a Boolean expression that can contain a coverage item name and constants.</p> <p>The Boolean expression is evaluated in a global context, not in instances of the struct, i.e., the expression shall be valid at all times, even before generation. Therefore, only constants and the item itself can be used in the expression. For example, if <i>i</i> is a coverage item and <i>j</i> is a reference to a struct field, the expression <i>i</i> &gt; 5 is a valid expression, but not <i>i</i> &gt; me . j.</p> <p>If the <b>ignore</b> expression is TRUE when the data is sampled, the sampled value is ignored (not added to the bucket count).</p> <p>To achieve the first effect (ignore specific samples) without hiding buckets containing holes (and to have the grade reflect all generated values), use the <b>when</b> option instead.</p>
<b>illegal</b> =item-bool-exp	<p>Defines values that are illegal. An illegal value shall cause a DUT error. If the <b>check_illegal_immediately</b> coverage configuration option is FALSE, the DUT error occurs during the <b>check_test</b> phase of the test. If that configuration option is TRUE, the DUT error occurs immediately (on-the-fly).</p> <p>The Boolean expression is evaluated in a global context, not in instances of the struct, i.e., the expression shall be valid at all times, even before generation. Therefore, only constants and the item itself can be used in the expression. For example, if <i>i</i> is a coverage item and <i>j</i> is a reference to a struct field, the expression <i>i</i> &gt; 5 is a valid expression, but not <i>i</i> &gt; me . j.</p> <p>If the coverage grades need to reflect all bucket contents, use the <b>when</b> option instead to specify the circumstances under which a given value is counted.</p>



Table 32—Coverage item options (*continued*)

Option	Description
<b>radix=DEC   HEX   BIN</b>	For items of type <b>int</b> or <b>uint</b> , this specifies the radix used in coverage reports for implicit buckets. If the <b>ranges</b> option is not used to create explicit buckets for an item, a bucket is created for every value of the item that occurs in the test. Each different value sampled gets its own bucket, with the value as the name of the bucket. These are called <i>implicit buckets</i> . The global <b>print</b> radix option does not affect the bucket name radix. Legal values are <b>DEC</b> (decimal), <b>HEX</b> (hexadecimal), and <b>BIN</b> (binary). The value shall be in uppercase letters. If the <b>radix</b> is not used, <b>int</b> or <b>uint</b> bucket names are displayed in decimal. If no radix is specified for an item, but a radix is specified for the item's group, the group's radix applies to the item.
<b>weight=uint</b>	Specifies the weight of the current item relative to other items in the same coverage group. It is a non-negative integer with a default of 1.

This defines a new basic coverage item with an optional type. Options specify how coverage data is collected and reported for the item. The item can be an existing field name or a new name. If a new name is used for a coverage item, the item's type and the expression that defines it shall also be specified.

If a value for an item falls outside all of the buckets for the item, that value does not count toward the item's grade. The **ranges** option determines the number and size of buckets into which values for the item are placed. If **ranges** is not specified, the default number of buckets is 16 [set by the **max\_int\_buckets** coverage configuration option (see 29.9)]. The **ranges** option must be used for items of **real** type, in which case the *range* and *every-count* parameters may be specified in constant real values. If buckets are not created for all possible values of the item, the values for which buckets do not exist are ungradeable. Those values are given goals of 0 and do not affect the grade for the item.

For example, a randomly generated item of type **uint** has  $2^{32}$  possible values. If no ranges are specified for a **uint** item, then buckets are created by default for only the first 16 possible values (0 through 15). Since the odds that a **uint** value will be less than 16 are very small, it is almost certain that none of the values will fall into one of the 0 to 15 buckets, which are the only buckets for which a grade is calculated. This means that the item does not receive a grade or contribute to the grade for the group.

NOTE—Unless coverage mode is turned on first (see 29.9), no coverage results are collected, even if cover groups and cover items are defined.

Syntax example:

```
cover inst_driven is {
  item op1;
  item op2;
  item op2_big : bool = (op2 >= 64);
  item hdl_sig : int  = 'top.sig_1'
}
```

### 15.2.1 Coverage per instance

The coverage per instance feature (the **per\_instance** option) enables the collection of coverage information for separate instances of structs or units, and the verification of the coverage data and grade associated with each particular instance.

When the **per\_instance** option is set in a cover item definition, that item becomes a “**per\_instance** item.” Each bucket of that item gets its own coverage grade and is shown separately in the coverage report. An

instance is created for every valid bucket of the **per\_instance** item. Any instance that is not sampled is marked as a hole. For example, if a struct has a field named `packet_type` and the value of the `packet_type` field can be either `Ethernet` or `ATM`, then making that field a **per\_instance** item results in a grade and a coverage report listing for `Ethernet` instances, and a separate grade and coverage report listing for `ATM` instances.

Along with the **per\_instance** item data, coverage data is also collected for the original **per\_type** item as if it were not a **per\_instance** item. This coverage data for the **per\_type** item is the accumulated information for all the instances, using the coverage options defined for the item.

Grading is calculated for each instance separately. The grade of the cover group is the weighted grades of all the **per\_instance** items. The **per\_type** item receives the same grade it would get if there were no **per\_instance** items.

An instance item name is the name of the **per\_type** item followed by `==` and the name of the instance bucket. For example, the instance item names for the preceding case are:

```
packet_type == Ethernet
packet_type == ATM
```

The following considerations also apply:

- For integer instances, the *decimal radix* is used regardless of what the radix is for the cover group.
- More than one **per\_instance** item can be defined in the same cover group. In this case, the total number of instances is the sum of all valid buckets for all the **per\_instance** items +1 (the **per\_type** bucket).
- If a **per\_instance** item definition is changed in an extension, then the coverage data for the original **per\_type** item might not accurately reflect nor agree with the coverage data collected per instance.
- A **per\_instance** item cannot be defined under a specific instance.
- Items with the same name can be defined under two different instances, as long as they have the same definition (type and expression).
- If a **per\_instance** item is participating in a cross item or a transition item, then the cross or transition item is not added to the instances created by the **per\_instance** item.
- To cancel per instance coverage collection in an extension, use the **also per\_instance = FALSE** option.

### 15.2.2 per\_instance item errors

[Table 33](#) lists errors that can occur when coverage per instance is used.

**Table 33—Coverage per instance errors**

Error	Description
Using a non-gradeable item as a <b>per_instance</b> item	When the user defines a <b>per_instance</b> item option for a non-gradeable item, a runtime error shall be issued.
Using a cross or transition item as a <b>per_instance</b> item	When the user defines a <b>per_instance</b> item for a cross item or a transition item, a loadtime error shall be issued.
Trying to extend an invalid instance	When the user tries to extend (using <b>cover ... is also</b> ) a group instance that does not exist, a loadtime error shall be issued.

**Table 33—Coverage per instance errors (*continued*)**

Error	Description
Recursively split instances	When the user defines a <b>per_instance</b> item option for an instance group extension, a loadtime error shall be issued.
Trying to extend specific instances without using <b>is also</b>	When the user tries to extend a specific group instance using <b>is</b> instead of <b>is also</b> .
Trying to define multiple items with the same name but different definitions under different instances	When the user tries to define an item with the same name under two different instances.
Specifying an invalid instance name	When the user specifies an invalid instance name (possibly by using wild cards). No matching instance is found and the command is ignored.

### 15.3 Defining cross coverage items: cross

<b>Purpose</b>	Define a cross coverage item
<b>Category</b>	Coverage group item
<b>Syntax</b>	<b>cross</b> <i>item-name-1</i> , <i>item-name-2</i> , ... [ <b>using</b> <i>coverage-item-option</i> , ...]
<b>Parameters</b>	<i>item-name-1</i> , <i>item-name-2</i> <p>Each item name shall be one of the following:</p> <ul style="list-style-type: none"> <li>— The name of an item defined previously in the current coverage group.</li> <li>— The name of a transition item defined previously in the current coverage group</li> <li>— The name of a cross item defined previously in the current coverage group.</li> </ul>
	<i>coverage-item-option</i> <p>An option for the cross item (see <a href="#">Table 34</a>).</p>

**Table 34—Cross coverage item options**

Option	Description
<b>name</b> = <i>label</i>	Specifies a name for a cross coverage item. No white spaces are allowed in the label. The default is <code>cross__item-name-1__item-name-2</code> .
<b>text</b> = <i>string</i>	A text description for this coverage item. This can only be a quoted string (" "), not a variable or expression. The text is shown along with the item name at the top of the coverage information for the item.
<b>when</b> = <i>bool-exp</i>	The item is sampled only when <i>bool-exp</i> is TRUE. The <i>bool-exp</i> is evaluated in the context of the parent struct, i.e., the scope of entities is taken from that context. Concretely, names of fields shall be taken as fields of the parent structs, and methods called shall be executed in the context of said struct. The pseudo-variable <b>me</b> refers to that struct.

Table 34—Cross coverage item options (*continued*)

Option	Description
<b>no_collect</b>	This cross coverage item is not displayed in coverage reports and is not saved in the coverage files. However, it can be referenced by other cross and transition items.
<b>at_least=num</b>	The minimum number of samples for each bucket of the item. Anything less than <i>num</i> is considered a hole. This option cannot be used with string items or for unconstrained integer items (items that do not have specified ranges). This shall be a non-negative number; the default is 1.
<b>ignore=item-bool-exp</b>	Defines values that are to be completely ignored. They do not appear in the statistics at all. The expression is a Boolean expression that can contain a coverage item name and constants. The Boolean expression is evaluated in a global context, not in instances of the struct, i.e., the expression shall be valid at all times, even before generation. Therefore, only constants and the item itself can be used in the expression. In a cross, this means any of the participating items. In a transition, it means the item or <b>prev_item</b> . For example, if <i>i</i> is a coverage item and <i>j</i> is a reference to a struct field, the expression <i>i</i> > 5 is a valid expression, but not <i>i</i> > me.j. If the <b>ignore</b> expression is TRUE when the data is sampled, the sampled value is ignored (not added to the bucket count). To achieve the first effect (ignore specific samples) without hiding buckets containing holes (and to have the grade reflect all generated values), use the <b>when</b> option instead.
<b>illegal=item-bool-exp</b>	Defines values that are illegal. An illegal value shall cause a DUT error. If the <b>check_illegal_immediately</b> coverage configuration option is FALSE, the DUT error occurs during the <b>check_test</b> phase of the test. If that configuration option is TRUE, the DUT error occurs immediately (on-the-fly). The Boolean expression is evaluated in a global context, not in instances of the struct, i.e., the expression shall be valid at all times, even before generation. Therefore, only constants and the item itself can be used in the expression. In a cross, this means any of the participating items. For example, if <i>i</i> is a coverage item and <i>j</i> is a reference to a struct field, the expression <i>i</i> > 5 is a valid expression, but not <i>i</i> > me.j. If the coverage grades need to reflect all bucket contents, use the <b>when</b> option instead to specify the circumstances under which a given value is counted.
<b>weight=uint</b>	Specifies the weight of the current cross item relative to other items in the same coverage group. It is a non-negative integer with a default of 1.

This defines cross coverage between items in the same coverage group. It creates a new item with a name specified using a **name** option or by using the default name of `cross__item-name-1__item-name-2` (with two underscores separating the parts of the name). This shows every combination of values of the first and second items, and every combination of the third item and the first item, the third item and the second item, and so on. Any combination of basic coverage items, cross items, and transitions defined in the same coverage group can be crossed.

The **using when**, **using ignore**, and **using illegal** options of the constituent items restrict the sampled values for a cross item. For example, in the `inst_driven` coverage group below, if `item opcode` includes the option `using ignore = (opcode != ADD)`, the cross coverage item `cross__opcode__op1` would exclude all buckets with the `opcode` value `ADD`.

Syntax example:

```
cover inst_driven is {
    item opcode;
    item op1;
    cross opcode, op1
}
```

## 15.4 Defining transition coverage items: transition

<b>Purpose</b>	Define a coverage transition item
<b>Category</b>	Coverage group item
<b>Syntax</b>	<b>transition</b> <i>item-name</i> [ <b>using</b> <i>coverage-item-option</i> , ...]
<b>Parameters</b>	<i>item-name</i> A coverage item defined previously in the current coverage group.
	<i>coverage-item-option</i> An option for the transition item (see <a href="#">Table 35</a> ).

This defines coverage for changes from one value to another of a coverage item. If no name is specified for the transition item with the **name** option, it gets a default name of `transition__item-name` (with two underscores between `transition` and *item-name*). If *item-name* had *n* samples during the test, then the transition item has *n*–1 samples, where each sample has the format *previous-value*, *value*.

Syntax example:

```
cover state_change is {
    item st : cpu_state = 'top.cpu.main_cur_state';
    transition st
}
```

**Table 35—Transition coverage item options**

Option	Description
<b>name</b> = <i>string</i>	Specifies a name for a transition coverage item. The default name is <code>transition__item-name</code> (where two underscores separate <code>transition</code> and <i>item-name</i> ).
<b>text</b> = <i>string</i>	A text description for this coverage item. This can only be a quoted string (" "), not a variable or expression. The text is shown along with the item name at the top of the coverage information for the item.
<b>no_collect</b>	This transition coverage item is not displayed in coverage reports and is not saved in the coverage files. However, it can be referenced by other cross and transition items.
<b>when</b> = <i>bool-exp</i>	The item is sampled only when <i>bool-exp</i> is TRUE. The <i>bool-exp</i> is evaluated in the context of the parent struct, i.e., the scope of entities is taken from that context. Concretely, names of fields shall be taken as fields of the parent structs, and methods called shall be executed in the context of said struct. The pseudo-variable <b>me</b> refers to that struct.
<b>at_least</b> = <i>num</i>	The minimum number of samples for each bucket of each of the transition items. Anything less than <i>num</i> is considered a hole. This option cannot be used with string items or for unconstrained integer items (items that do not have specified ranges). This shall be a non-negative number; the default is 1.

**Table 35—Transition coverage item options (continued)**

Option	Description
<b>ignore</b> = <i>item-bool-exp</i>	<p>Define values that are to be completely ignored. They do not appear in the statistics at all. The expression is a Boolean expression that can contain a coverage item name and constants. The previous value can be accessed as <b>prev</b>_<i>item-name</i>. The <b>prev</b> prefix is predefined for this purpose.</p> <p>The Boolean expression is evaluated in a global context, not in instances of the struct, i.e., the expression shall be valid at all times, even before generation. Therefore, only constants and the item itself can be used in the expression. In a cross, this means any of the participating items. In a transition, it means the item or <b>prev</b>_<i>item</i>. For example, if <i>i</i> is a coverage item and <i>j</i> is a reference to a struct field, the expression <i>i</i> &gt; 5 is a valid expression, but not <i>i</i> &gt; me.<i>j</i>.</p> <p>If the <b>ignore</b> expression is TRUE when the data is sampled, the sampled value is ignored (not added to the bucket count).</p> <p>To achieve the first effect (ignore specific samples) without hiding buckets containing holes (and to have the grade reflect all generated values), use the <b>when</b> option instead.</p>
<b>illegal</b> = <i>item-bool-exp</i>	<p>Define values that are illegal. An illegal value shall cause a DUT error. If the <b>check_illegal_immediately</b> coverage configuration option is FALSE, the DUT error occurs during the <b>check_test</b> phase of the test. If that configuration option is TRUE, the DUT error occurs immediately (on-the-fly).</p> <p>The Boolean expression is evaluated in a global context, not in instances of the struct, i.e., the expression shall be valid at all times, even before generation. Therefore, only constants and the item itself can be used in the expression. In a transition, this means any of the participating items. For example, if <i>i</i> is a coverage item and <i>j</i> is a reference to a struct field, the expression <i>i</i> &gt; 5 is a valid expression, but not <i>i</i> &gt; me.<i>j</i>.</p> <p>If the coverage grades need to reflect all bucket contents, use the <b>when</b> option instead to specify the circumstances under which a given value is counted.</p>
<b>weight</b> = <i>uint</i>	<p>Specifies the weight of the current transition item relative to other items in the same coverage group. It is a non-negative integer with a default of 1.</p>

## 15.5 Extending coverage groups: cover ... using also ... is also

<b>Purpose</b>	Extend a coverage group
<b>Category</b>	Struct member
<b>Syntax</b>	<b>cover</b> <i>event-type</i> <b>using also</b> <i>cover-option</i> , ...[ <b>is also</b> { <i>coverage-item-definition</i> ; ... } ]
<b>Parameters</b>	<i>event-type</i> The name of the coverage group. This shall be an event defined previously in the struct. The event is the sampling event for the coverage group.
	<i>cover-option</i> The definition for this option is shown in <a href="#">Table 32</a> .
	<i>coverage-item-definition</i> The definition of a coverage item (see <a href="#">Table 31</a> ).

The **using also** clause changes, overrides, or extends options previously defined for the coverage group. The **is also** clause adds new items to a previously defined coverage group, or it can be used to change the options for previously defined items (see [15.6](#)).

The following considerations also apply:

- If a coverage group is defined under a **when** subtype, it can only be extended under that subtype.

- If **per\_instance** coverage is being used (see [15.2.1](#)), a particular cover group instance can be extended to complement or override options set in the base type cover group. To change an item's options in a particular instance, enter the instance name in the **cover is also** construct.
- If an instance is extended, but it never gets created (due to an **ignore** or **illegal** option), a warning is issued and no information for the extension is put in the coverage data.
- If the coverage options of an instance are changed, the coverage data for the **per\_type** item might no longer reflect or agree with the **per\_instance** coverage data.
- If, in an extension of a cover group, a cover group **when** option is overridden, then the overriding condition is only considered after the condition in the base group is satisfied, i.e., sampling of the item is only performed when the logical AND of the cover group **when** options are TRUE.
- When **using also** is used to extend or change a **when**, **illegal**, or **ignore** option, a special variable named **prev** is automatically created. The **prev** variable holds the results of all previous **when**, **illegal**, or **ignore** options, so it can be used as a shorthand to assert those previous options combined with a new option value.

Syntax examples:

```
cover rclk is also {
    item rflag
};
cover rclk using also text = "RX clock";
cover rclk using also no_collect is also {
    item rvalue
}
```

## 15.6 Extending coverage items: item ... using also

<b>Purpose</b>	Change or extend the options on a cover item	
<b>Category</b>	Coverage group item	
<b>Syntax</b>	<b>item</b> <i>item-name</i> <b>using also</b> <i>coverage-item-option</i> , ...	
<b>Parameters</b>	<i>item-name</i>	The name assigned to the coverage item. If the optional <i>type=exp</i> is not used, the value of the field named <i>item-name</i> is used as the coverage sample value.
	<i>coverage-item-option</i>	The coverage item options are listed in <a href="#">Table 32</a> . The options can appear in any order after the <b>using</b> keyword.

Cover item extensibility enables extending, changing, or overriding a previously defined coverage item. To extend a coverage item, see [15.5](#).

If a coverage item is originally defined under a **when** subtype, it can only be extended in the same subtype of the base type.

When **using also** is used to extend or change a **when**, **illegal**, or **ignore** option, a special variable named **prev** is automatically created. The **prev** variable holds the results of all previous **when**, **illegal**, or **ignore** options, so it can be used as a shorthand to assert those previous options combined with a new option value.

Once an item is extended, it shall be referenced using its full name. If an item with that name does not exist, an error shall be issued.

Syntax example:

```
item len using also radix = HEX
```

## 15.7 Coverage API

This subclause contains descriptions of the *e* coverage API.

The coverage API is accessed through a set of methods defined in the struct **user\_cover\_struct**. Once an instance of this struct is declared, the **scan\_cover()** method may be invoked which, in turn, invokes accessory methods, such as **start\_group()** and **start\_instance()**.

### 15.7.1 Methods of user\_cover\_struct

This subclause contains descriptions of the *e* coverage predefined API methods.

#### 15.7.1.1 scan\_cover()

<b>Purpose</b>	Activate the coverage API and specify items to cover
<b>Category</b>	Method
<b>Syntax</b>	<b>scan_cover</b> ( <i>item-names</i> :string): int
<b>Parameters</b>	<i>item-names</i> The names of the coverage items to be scanned by <b>scan_cover()</b> . This is a string of the form <i>struct-name.group-name.item-name</i> (for example, "inst.start.opcode"). Wild cards are allowed.

The **scan\_cover()** method initiates the coverage data-scanning process. It goes through all the items in all the groups specified in the *item-names* parameter in the order that groups and items have been defined. This method cannot be extended.

For each group, **scan\_cover()** calls **start\_group()**. For each instance in the group, **scan\_cover()** calls **start\_instance()**. For each item in the current instance, **scan\_cover()** calls **start\_item()**. Then for each bucket of the item, **scan\_cover()** calls **scan\_bucket()**. After all of the buckets of the item have been processed, **scan\_cover()** calls **end\_item()**. After all items of the instance have been processed, **scan\_cover()** calls **end\_instance()**. After all instances in the group have been processed, **scan\_cover()** calls **end\_group()**.

Before each call to any of the preceding methods, the relevant fields in the **user\_cover\_struct** are updated to reflect the current item [and also the current bucket for **scan\_bucket()**].

The **scan\_cover()** method returns the number of coverage items actually scanned.

NOTE—The methods called by **scan\_cover()**—**start\_group()**, **start\_instance()**, **start\_item()**, **scan\_bucket()**, **end\_item()**, **end\_instance()**, and **end\_group()**—are initially empty and meant to be extended.

Syntax example:

```
num_items = cover_info.scan_cover("cpu.inst_driven.*")
```



### 15.7.1.2 start\_group()

<b>Purpose</b>	Process coverage group information according to user preferences
<b>Category</b>	Method
<b>Syntax</b>	<b>start_group()</b>

When the **scan\_cover()** method initiates the coverage data-scanning process for a group, it updates the group-related fields within the containing **user\_cover\_struct** and then calls the **start\_group()** method. The **start\_group()** method is called for every group to be processed by **scan\_cover()**. For every instance within a group, **scan\_cover()** calls the **start\_instance()** method.

The **start\_group()** method is originally empty. It is meant to be extended to process group data according to user preferences.

NOTE—**start\_group()**, **start\_instance()**, and **scan\_cover()** are all methods of the **user\_cover\_struct**.

Syntax example:

```
start_group() is {
  if group_text != NULL then {
    out("Description: ", group_text)
  }
}
```

### 15.7.1.3 start\_instance()

<b>Purpose</b>	Process coverage instance information according to user preferences
<b>Category</b>	Method
<b>Syntax</b>	<b>start_instance()</b>

When the **scan\_cover()** method initiates the coverage data-scanning process for an instance, it updates the instance-related fields within the containing **user\_cover\_struct** and then calls the **start\_instance()** method. The **start\_instance()** method is called for every instance to be processed by **scan\_cover()**.

The **start\_instance()** method is originally empty. It is meant to be extended to process instance data according to user preferences.

NOTE—**start\_instance()** and **scan\_cover()** are methods of the **user\_cover\_struct**.

Syntax example:

```
start_instance() is {
  if instance_text != NULL then {
    out("Description: ", instance_text)
  }
}
```

#### 15.7.1.4 start\_item()

<b>Purpose</b>	Process coverage item information according to user preferences
<b>Category</b>	Method
<b>Syntax</b>	<b>start_item()</b>

When the **scan\_cover()** method initiates the coverage data-scanning process for an item, it updates the item-related fields within the containing **user\_cover\_struct** and then calls the **start\_item()** method. The **start\_item()** method is called for every item to be processed by **scan\_cover()**.

The **start\_item()** method is originally empty. It is meant to be extended to process item data according to user preferences.

NOTE—**start\_item()** and **scan\_cover()** are methods of the **user\_cover\_struct**.

Syntax example:

```
start_item() is {
    if item_text != NULL then {
        out("Description: ", item_text)
    }
}
```

#### 15.7.1.5 scan\_bucket()

<b>Purpose</b>	Process coverage item information according to user preferences
<b>Category</b>	Method
<b>Syntax</b>	<b>scan_bucket()</b>

When the **scan\_cover()** method processes coverage data, then for every bucket of the item, it updates the bucket-related fields within the containing **user\_cover\_struct** and calls **scan\_bucket()**.

The **scan\_bucket()** method is originally empty. It is meant to be extended to process bucket data according to user preferences.

NOTE—**scan\_bucket()** and **scan\_cover()** are methods of the **user\_cover\_struct**.

Syntax example:

```
scan_bucket() is {
    out(count, " ", percent, "% ", bucket_name)
}
```

### 15.7.1.6 end\_item()

<b>Purpose</b>	Report end of item coverage information according to user preferences
<b>Category</b>	Method
<b>Syntax</b>	<b>end_item()</b>

When the **scan\_cover()** method completes the processing of coverage data for an item, it calls the **end\_item()** method to report the end of item information according to user preferences. When all items in the current group have been processed, **scan\_cover()** calls the **start\_instance()** method for the next instance.

The **end\_item()** method is originally empty. It is meant to be extended so as to process item data according to user preferences.

NOTE—**end\_item()**, **start\_instance()**, and **scan\_cover()** are all methods of the **user\_cover\_struct**.

Syntax example:

```
end_item() is {
    out("finished item ", item_name, "\n")
}
```

### 15.7.1.7 end\_instance()

<b>Purpose</b>	Process end of instance coverage information according to user preferences
<b>Category</b>	Method
<b>Syntax</b>	<b>end_instance()</b>

When the **scan\_cover()** method completes the processing of coverage data for an instance, it calls the **end\_instance()** method to report the end of instance information according to user preferences. When all instances in the current group have been processed, **scan\_cover()** calls the **start\_group()** method for the next group.

The **end\_instance()** method is originally empty. It is meant to be extended so as to process instance data according to user preferences.

NOTE—**end\_instance()**, **start\_group()**, and **scan\_cover()** are all methods of the **user\_cover\_struct**.

Syntax example:

```
end_instance() is {
    out("finished instance ", instance_name, "\n")
}
```

### 15.7.1.8 end\_group()

<b>Purpose</b>	Report end of group coverage information according to user preferences
<b>Category</b>	Method
<b>Syntax</b>	<b>end_group()</b>

When the **scan\_cover()** method completes the processing of coverage data for a group, it calls the **end\_group()** method to report the end of group information according to user preferences.

The **end\_group()** method is originally empty. It is meant to be extended so as to process item data according to user preferences.

NOTE—**end\_group()** and **scan\_cover()** are both methods of the **user\_cover\_struct**.

Syntax example:

```
end_group() is {
    out("finished group", group_name, "\n")
}
```

### 15.7.2 Fields of user\_cover\_struct

In addition to the methods of **user\_cover\_struct** in [15.7.1](#), the **user\_cover\_struct** contains fields for coverage information. The fields are listed in [Table 36](#). The code below is referred to in the Description section of the table.

```
struct packet {
    kind: [tx, rx];
    len: uint (bits:7);
    keep len in [64..128];
    event done;
    cover done is {
        item kind;
        item len using ranges={range([0..127], "", 16)};
        transition len;
        cross kind, len
    }
}
```

**Table 36—Fields in user\_cover\_struct**

Field Name	Type	Description
<b>Group Information Fields</b>		
struct_name	string	The struct in which the item resides (for example, packet)
package_name	string	The name of the package in which the struct defining the items was declared. Syntax example: package_name : "Turbo Mode Package"

**Table 36—Fields in `user_cover_struct`**

Field Name	Type	Description
<code>group_name</code>	string	The name of the group (for example, done)
<code>group_text</code>	string	The text specified using the <b>text=</b> option in the group definition, or NULL if the option is not used
<code>group_weight</code>	int	The group's weight for grading
<code>group_grade</code>	int	The group's grade
NOTE—Group information fields are updated before <b>start_item()</b> is called for the first item in the group.		
<b>Per Instance Item Information Field</b>		
<code>instance_name</code>	string	Modified whenever a new instance is scanned.
<code>instance_grade</code>	string	If per instance coverage is used, then this is the field to look at, rather than <code>group_grade</code> .
<b>Item Information Fields</b>		
<code>item_name</code>	string	The name of the current item (for example, kind)
<code>sub_items</code>	list of string	For a basic item, the item name (for example {kind}) For a cross, the names of the items that make up the cross (for example {kind; len}) For a transition, the item name twice (for example {len; len})
<code>samples</code>	int	The total number of samples for the item in the .ecov files
<code>item_at_least</code>	int	The value of the <b>at_least</b> option in the item definition. This is the grading goal for the item, which is the minimum number of samples required to fill each bucket of the item.
<code>item_text</code>	string	The text specified using the <b>text=</b> option in the item definition, or NULL if the option is not used.
<code>item_no_of_tests</code>	int	The number of tests in which the item appears
<code>item_when_string</code>	string	The struct subtype under which the item is defined (for example, if the item was defined inside a “when tx packet”, then the <code>item_when_string</code> is “tx packet”).
<code>item_exp_string</code>	string	The expression string, if the item was defined using <i>type=exp</i> , or NULL if <i>type=exp</i> was not specified (for example, if the item was defined as “item s: int=sys.packets.size()”, then the <code>item_exp_string</code> is <code>sys.packets.size()</code> ).
<code>item_weight</code>	int	The item's weight for grading <code>item_grade</code> int The item's grade
<b>Bucket Information Fields</b>		
<code>bucket_name</code>	string	The name of the current bucket
<code>count</code>	int	The number of samples in this bucket status <code>bucket_status</code> Whether bucket is <b>illegal</b> , <b>normal</b> , or <b>hole</b> .
<code>at_least</code>	int	The number of samples required to fill the bucket
<code>no_of_tests</code>	int	The number of tests that sample the bucket
<code>cross_level</code>	int	For cross coverage, ranging from 0 to <code>sub_items.size()-1</code>
<code>is_string</code>	bool	TRUE if this is a string bucket. If TRUE, the string is the bucket name

**Table 36—Fields in `user_cover_struct`**

Field Name	Type	Description
<code>int_value</code>	int	Meaningful if <b>is_string</b> is FALSE, and the value in this bucket as an integer. If the bucket is a range of values, this is the lowest integer value within the range.
<code>hi_int_value</code>	int	Meaningful if <b>is_string</b> is FALSE, and the value in this bucket is an integer. If the bucket is a range of values, this is the highest integer value within the range.
<code>bucket_weight</code>	int	The bucket's weight for grading <code>bucket_grade</code> int The bucket's grade full_bucket_name list of string A list of all of the bucket sets in the coverage hierarchy. This information is needed for grading cross items and transition items. The last element in the list is the name of the current bucket set or bucket.
<b>Overall Information Field</b>		
<code>all_grade</code>	int	The overall grade for all coverage items, calculated by calling the <b>scan_cover()</b> method with <code>*.*.*</code> as the <i>item</i> parameter. The default is UNDEF.

**Notes**

- Item information fields are updated before `start_item()` is called.
- Bucket information fields are updated before `scan_bucket()` is called.
- A grade is an integer from 0 to 100,000,000, or UNDEF for an ungradeable item. A grade of 100,000,000 means the goal for the number of samples has been reached. To represent the grade as a percentage, divide the grade by 1,000,000. Then, a grade of 100% means the goal has been reached, and a grade of 0 means no samples were collected.

**15.8 Coverage methods for the `covers` struct**

The **covers** struct is a predefined struct containing methods used for coverage and coverage grading. With the exception of the **write\_cover\_file()** method, all of the following methods are methods of the **covers** struct. See also [17.2.5](#).

**15.8.1 `include_tests()`**

<b>Purpose</b>	Specify for which tests coverage information should be displayed	
<b>Category</b>	Predefined method	
<b>Syntax</b>	<b><code>covers.include_tests</code></b> ( <i>full-run-name</i> : string, <i>include-run</i> : bool)	
<b>Parameters</b>	<i>full-run-name</i>	The name of the test to include or exclude.
	<i>include-run</i>	Set to TRUE to include the specified run, FALSE to exclude it.

This method specifies which test runs to use in showing coverage information. If `.ecov` files are being read to load coverage information, only call this method after the `.ecov` files have been read.

Syntax example:

```
covers.include_tests("tests_A:run_A_10", TRUE)
```

### 15.8.2 set\_weight()

<b>Purpose</b>	Specify the coverage grading weight of a group or item	
<b>Category</b>	Predefined method	
<b>Syntax</b>	<b>covers.set_weight</b> ( <i>entity-name</i> : string, <i>value</i> : int, <i>multiply-value</i> : bool)	
<b>Parameters</b>	<i>entity-name</i>	The group or item for which to set the weight. This can include wild cards.
	<i>value</i>	The integer weight value to set.
	<i>multiply-value</i>	When this is FALSE, it changes the weights of all matching groups or items to <i>value</i> . When this is TRUE, the weights of all matching groups or items are multiplied by <i>value</i> .

Coverage grading uses weights to emphasize the affect of particular groups or items relative to others. The weights can be specified in the coverage group or item definitions. This method sets the weights procedurally. It overrides the weights set in the group or item definitions. Weights can be set explicitly or multiplied by a given value.

If .ecov files are being read to load coverage information, only call this method after the .ecov files have been read.

Syntax example:

```
covers.set_weight("inst.done", 4, FALSE)
```

### 15.8.3 set\_at\_least()

<b>Purpose</b>	Set the minimum number of samples needed to fill a bucket	
<b>Category</b>	Predefined method	
<b>Syntax</b>	<b>covers.set_at_least</b> ( <i>entity-name</i> : string, <i>value</i> : int, <i>multiply-value</i> : bool)	
<b>Parameters</b>	<i>entity-name</i>	The group or item for which to set the “at-least” number. This can include wild cards.
	<i>value</i>	The at-least integer value to set.
	<i>multiply-value</i>	When this is FALSE, it changes the at-least number of all matching items to <i>value</i> . When this is TRUE, it multiplies the at-least number by <i>value</i> .

The minimum number of samples required to fill a bucket can be set in the coverage group or item definitions. This method can be used to set the number procedurally. It overrides the numbers set in the group or item definitions. If the *entity-name* is a coverage group name, all items in the group are affected. If the *entity-name* matches items within a coverage group, only those items are affected.

If .ecov files are being read to load coverage information, only call this method after the .ecov files have been read.

Syntax example:

```
covers.set_at_least("inst.done", 4, FALSE)
```

#### 15.8.4 set\_cover()

<b>Purpose</b>	Turn coverage data collection and display on or off for specified items or events	
<b>Category</b>	Predefined method	
<b>Syntax</b>	<b>covers.set_cover</b> (( <i>item</i>   <i>event</i> ): string, <i>collect-coverage</i> : exp)	
<b>Parameters</b>	<i>item</i>	A string, enclosed in double quotes (" "), specifying the coverage item to turn on or off. This can include wild cards.
	<i>event</i>	<p>A string, enclosed in double quotes (" "), specifying the event to turn on or off. This can include wild cards.</p> <p>Enter the name of the event using the following syntax:</p> <p style="text-align: center;"><b>session.events.struct_type__event_name</b></p> <p>where <i>struct_type</i> and <i>event_name</i> are separated by two underscores. Wild cards can also be used here.</p> <p>If only one name is specified, it is treated as a struct type and the method shall affect all events in that struct type.</p>
	<i>collect-coverage</i>	Set to TRUE to turn on coverage for the item or FALSE to turn coverage off.

By default, coverage data is collected for all defined coverage items and groups, and for all user-defined events. This method selectively turns data collection on or off for specified items, groups, or events. Although this method can be used to filter samples during periods in which they are not valid, for performance reasons, use **when** subtypes instead.

Additionally, if the test ends while coverage collection is turned off by **set\_cover()** for one or more coverage groups, then **set\_cover()** needs to be called again to re-enable sampling before the .ecov file is written, in order to include the previously collected samples for those groups in the .ecov file.

Syntax example:

```
covers.set_cover("packet.*", FALSE)
```

#### 15.8.5 get\_contributing\_runs()

<b>Purpose</b>	Return a list of the test runs that contributed samples to a bucket	
<b>Category</b>	Predefined method	
<b>Syntax</b>	<b>covers.get_contributing_runs</b> ( <i>item-name</i> : string, <i>bucket-name</i> : string): list of string	
<b>Parameters</b>	<i>item-name</i>	A string, enclosed in double quotes (" "), specifying the coverage item that contains <i>bucket-name</i> .
	<i>bucket-name</i>	A string, enclosed in double quotes (" "), specifying the bucket for which contributing test run names are to be listed.

This method returns a list of strings that are the full run names of the test runs that placed samples in a specified bucket. For a cross item, the *bucket-name* can be a bucket of any level, with the bucket set names separated by slashes, e.g., ADD/REG1/[0xC0..0xCF].



Syntax example:

```
bkl = covers.get_contributing_runs("inst.done.len", "[0..4]")
```

### 15.8.6 get\_unique\_buckets()

<b>Purpose</b>	Return a list of the names of unique buckets from specific tests.
<b>Category</b>	Method
<b>Syntax</b>	<b><code>covers.get_unique_buckets</code></b> ( <i>file-name</i> : string): list of string
<b>Parameters</b>	<i>file-name</i> A string, enclosed in double quotes, ( " ") specifying which coverage database files, containing unique buckets, to display. Wild cards cannot be used in the file name.

A *unique bucket* is a bucket that is covered by only one test. This method reports, for each specified test, the full names of its unique buckets, if there are any.

Syntax example:

```
print covers.get_unique_buckets("test_rx")
```

### 15.8.7 write\_cover\_file()

<b>Purpose</b>	Write the coverage results during a test
<b>Category</b>	Predefined method
<b>Syntax</b>	<b><code>write_cover_file</code></b> ()

This method writes the coverage results `.ecov` file during a test run. It can only be invoked during a test, not before the run starts nor after it ends.

The coverage file written by this method does not contain the `session.end_of_test` or `session.events` coverage groups.

Syntax example:

```
write_cover_file()
```

### 15.8.8 get\_overall\_grade()

<b>Purpose</b>	Return the normalized overall coverage grade
<b>Category</b>	Predefined method
<b>Syntax</b>	<b><code>covers.get_overall_grade</code></b> (): int

This method returns an integer that represents the overall coverage grade for the current coverage results. Since *e* does not handle floating point types, the value is a normalized value between 1 and 100M. To obtain a value equivalent to the overall grade, divide the returned value by 100M.

Syntax example:

```
grade = covers.get_overall_grade()
```

#### 15.8.9 get\_ecov\_name()

<b>Purpose</b>	Return the name of the .ecov file
<b>Category</b>	Predefined method
<b>Syntax</b>	<b><code>covers.get_ecov_name()</code></b> : string

This method returns the name of the .ecov file in which the current coverage results are to be stored.

Syntax example:

```
ecov_file = covers.get_ecov_name()
```

#### 15.8.10 get\_test\_name()

<b>Purpose</b>	Return the name of the current test
<b>Category</b>	Predefined method
<b>Syntax</b>	<b><code>covers.get_test_name()</code></b> : string

This method returns the identifier of the current test run.

Syntax example:

```
ecov_file = covers.get_test_name()
```

#### 15.8.11 get\_seed()

<b>Purpose</b>	Return the value of the seed for the current test
<b>Category</b>	Predefined method
<b>Syntax</b>	<b><code>covers.get_seed()</code></b> : int

This method returns the current test seed.

Syntax example:

```
seed_val= covers.get_seed()
```

## 16. Macros

*e* is an extensible language. Its syntax and lexicon can be extended using macros—declared by *define-as* and *define-as-computed* statements. Unlike macros in languages such as C, which are preprocessor-based, *e* macros are real syntactic rules applied by the parser just like those for built-in constructs. Macros are used to add new constructs to the language, possibly with new keywords and operators. The newly declared constructs are implemented by defining a syntactic expansion, reducing them to existing constructs.

### 16.1 Overview

#### 16.1.1 *e* syntactic structure

*e* syntax in general, and user-defined syntax (introduced by macros) in particular, is specified in terms similar to BNF. Each macro constitutes an alternative for the production of some non-terminal. But, there are a number of essential differences between the *e* grammar thus specified and the input of standard parser-generators, as follows:

- a) The terminals of the grammar are simply characters from the input (after trivial normalization and preprocessing). There are no separate lexical rules.
- b) The term sequence in production rules can be expressed using regular-expression-like operators, (akin to EBNF).
- c) There is a total precedence order over alternatives of a given non-terminal.

#### 16.1.2 Macro definition

The definition of a *macro* is analogous to that of a function. It consists of a header and a body. The header declares:

- a) The macro name
- b) The syntactic type or category that it returns
- c) The syntactic pattern of the new construct together with the syntactic arguments the macro takes. This part is called the *match expression*.

The body specifies what code is expanded once the macro is matched. The resulting expansion may be any *e* code that is appropriate in the context of the macro's syntactic category. Simple and computed macros differ in the way they define the rewrite rule:

- The rule for simple macros is given as a parameterized code segment (template) in which arguments taken from the input are embedded at specified locations.
- The rule for computed macros is given as an *e* action block that executes at parse time and procedurally computes the replacement code. Arguments taken from the input are available within this block as string variables.

## 16.2 define-as statement

<b>Purpose</b>	Define a new construct with parameterized code expansion	
<b>Category</b>	Statement	
<b>Syntax</b>	<b>define</b> <tag'non-terminal-type> "match-expression" <b>as</b> { replacement }	
<b>Parameters</b>	<i>tag</i>	An identifier that is used to name the macro. It needs to be unique in the scope of the non-terminal type. The name plays no actual role in the definition.
	<i>non-terminal-type</i>	A non-terminal of the <i>e</i> grammar corresponding to the syntactic category that is being extended by this macro (see 4.2). It must be one of <i>statement</i> , <i>struct_member</i> , <i>action</i> , <i>exp</i> , <i>type</i> , or <i>cover_item</i> .
	<i>match-expression</i>	A sequence of terminals and non-terminals that constitute a new choice for the derivation of the specified <i>non-terminal</i> (see 16.4).
	<i>replacement</i>	The parameterized replacement code to which the new construct is expanded. The expansion must constitute a legal <i>e</i> construct of the same category, or a sequence thereof (see 16.6).

Syntax example:

```
define <largest'action> "largest <exp> <num>" as {
    if <num> > <exp> then {<exp> = <num>}
}
```

## 16.3 define-as-computed statement

<b>Purpose</b>	Define a new construct by reduction	
<b>Category</b>	Statement	
<b>Syntax</b>	<b>define</b> <tag'non-terminal-type> "match-string" <b>as computed</b> { action;... }	
<b>Parameters</b>	<i>tag</i>	An identifier that is used to name the macro. It needs to be unique in the scope of the non-terminal type. The name plays no actual role in the definition.
	<i>non-terminal-type</i>	A non-terminal of the <i>e</i> grammar corresponding to the syntactic category that is being extended by this macro (see 4.2). It must be one of <i>statement</i> , <i>struct_member</i> , <i>action</i> , <i>exp</i> , <i>type</i> , or <i>cover_item</i> .
	<i>match-expression</i>	A sequence of terminals and non-terminals that constitute a new choice for the derivation of the specified <i>non-terminal</i> (see 16.4).
	<i>actions</i>	A block of actions that computes the expansion of the macro.

The block of *actions* is treated as the body of a method that returns a string. Thus, either the **result** variable or the **return** action should be used within it. The returned string must be a legal *e* construct of the same category, or a sequence thereof. (see 16.5).

The macro body is executed during parsing whenever the match expression matches the input text. Parsing of the entire module (or dependency unit in general, see [Annex B](#)) precedes semantic analysis for that module, so definitions occurring within the same module cannot be used or presupposed for purposes of the computation (even when they occur above the macro call).

Syntax example:

```
define <multi_field'struct_member> "[<MOD>private |protected ]<name>,... :
  <type>" as computed {
  for each (fname) in <names> do {
    result = append(result,<MOD>,fname,":",<type>,";");
  }
}
```

## 16.4 Match expression structure

The *match expression* consists of a sequence of terminals and non-terminals, possibly with regular expression operators such as alternative and optional sub-sequences.

### 16.4.1 Match expression terms

*Terminals* in the *e* grammar are simply ASCII characters. The grammar does not presuppose independent lexical analysis. The terminal part of a production is given literally in the match expression string. Some characters have special meaning in a match expression (see [16.4.2](#)), so they need to be escaped (\) to be taken literally as terminals.

The non-terminal types for user macros are the same ones available for extension: *statement*, *struct\_member*, *action*, *exp*, *type*, and *cover\_item*, which stand for a construct of the corresponding syntactic category (see [4.2](#)). A few “auxiliary” non-terminal types are also available: *name*, *num*, *file*, *block*, and *any*. An occurrence of a non-terminal in the match expression is marked by enclosing it with < and >. It consists of a non-terminal type selector, optionally preceded by an identifier serving as a tag (the format is <[tag']non-terminal-type>).

Non-terminals in the match expression declare formal syntactic arguments that can be used in the macro body. For example, the occurrence of <left'exp> inside a match expression is a declaration of a parameter by that name of the *exp* non-terminal type.

The role of the auxiliary non-terminals is shown in [Table 36](#). See also [4.1](#).

### 16.4.2 Match expression operators

The characters [, ], (, ), and | are used as the standard regular-expression operators — option, grouping, and alternation, respectively. These, along with < and > (marking the beginning and end of a syntactic argument), are not taken literally inside a match expression, unless they are escaped (\).

A restricted version of the repetition operator (akin to \* in standard regular-expression languages) is marked with ... (3 dots), preceded by a non-terminal specifier and a separator character. It is used to designate zero or more occurrences of the specified non-terminal separated by the designated character. For example, the syntax of the pseudo-routine **out()** is defined with the match-expression: "out(<exp>,...)".

NOTE—The character . (dot) does not need explicit escaping to be taken literally.

### 16.4.3 Submatches and labels

The part of the input text that is matched against the content of a grouping operator, an option operator, or a non-terminal upon the successful matching of a match expression is called a submatch. All submatches can be referenced inside the macro body.

A submatch may be given a label in the form *<label-tag>*, where *label-tag* is an identifier with capital letters and underscores only, immediately following the left parenthesis or left square bracket. For example, when the match expression "*(<WORD>Hello|Goodye) <name>*" is matched against the input "Goodbye John" the label *<WORD>* will hold the value "Goodbye"

**Table 36—Auxiliary non-terminals**

Parsing element	Description
<i>name</i>	A legal <i>e</i> name.
<i>num</i>	A literal numeric constant.
<i>file</i>	A UNIX-style file name.
<i>block</i>	A series of actions delimited by ; and enclosed between { }.
<i>any</i>	Any (possibly empty) sequence of characters from the input.

#### 16.4.4 Meta-grammar of match expression

The following grammar [usable in LALR (look-ahead, left-to-right) parsers] defines the syntax of match expressions described informally in [16.2](#) (non-literal terminals are in *italics*).

```

e_match_expression ::= sequence
                    | e_match_expression | sequence a
sequence ::= ε b
           | sequence term
           | sequence grouping
           | sequence option
           | sequence repetition
grouping ::= ( optional_label e_match_expression )
option  ::= [ optional_label e_match_expression ]
repetition ::= non_terminal character ...
term ::= literal
      | non_terminal
literal ::= character
        | literal character
non_terminal ::= < nt_selector >
              | < identifier ' nt_selector >
nt_selector ::= statement | struct_member | action | exp | type | name | num
              | block | file | any
optional_label ::= ε
               | < caps-identifier >

```

<sup>a</sup> In this expression, the second parallel bar ( | ) is a required literal symbol and not a list separator.

<sup>b</sup> This (term) denotes an empty string.

*identifier* is the standard *e* identifier and *character* is any ASCII character except the special ones: [ , ] , ( , ) , { , } , < , > , and \, or any one of these special characters when preceded by \.

#### 16.4.5 Proto-syntax

A number of characters have fixed syntactic function in *e*. They are [ , ] , ( , ) , { , } , and ". The syntactic role is reflected also in the structure of macro match expressions.

- Parentheses, square brackets, and curly brackets ([ , ] , ( , ) , { , } respectively) signify subordination of one construct to another. Within a match expression they need to be balanced, enclosing only a single non-terminal or a repetition thereof.
- Double quotes signify string literal whose content is opaque. They may enclose only the <any> non-terminal.
- The function of semicolon (;) is fixed to the sequencing of constructs inside curly brackets. It can be used in a match expression only as the separator of a non-terminal repetition inside curly brackets.

These restrictions can be expressed by adding the following rules to the grammar in [16.4.4](#):

```

term ::= literal
      | subform
      | bracketed_subform
bracketed_subform ::= \ ( subform \ )
                  | \ [ subform \ ]
                  | { subform }
                  | " <any> "
subform ::= non-terminal
        | repetition

```

Any use of ; (semicolon) in a match expression other than as a repetition separator character is illegal and so it is excluded from the *character* terminal specified in the grammar in [16.4.4](#).

## 16.5 Interpretation of match expressions

This subclause defines how interpretation of match expressions occurs.

### 16.5.1 Priority on production choices

A macro associates a new match choice with the non-terminal that is being extended. Each non-terminal already has a list of built-in production choices associated with it and might further have any number of production choices defined by previous macros. This does not limit the new match expression in any way and no ambiguity can be introduced thereby. The reason is that, unlike standard BNF interpretation, the productions are prioritized. Match expressions are tested according to their priority—the first one that succeeds gives the correct derivation for the input.

The priority is determined by the definition order of macros in the code—those whose definition appears later get higher priority (see [Annex B](#)). Thus, production choices declared by macros always have higher priority than built-in productions. In general, existing (built-in or user-defined) syntax can be overwritten by a new macro.

### 16.5.2 Recursive-decent interpretation

The modification to the grammar introduced by macros is best understood in terms of a recursive-decent backtracking parsing algorithm. Each non-terminal represents a routine that takes the string to be parsed as input and either succeeds in consuming some prefix of it or fails to do so. A non-terminal routine *succeeds* if at least one of the match expressions associated with it succeeds. The production that is actually used in this case is the production of highest priority that consumes the whole input. A match expression applies to the input if both its terminal parts match it and the substrings left for non-terminals succeed to be consumed by the corresponding non-terminal routine.

The non-terminal *fails* if no match expression associated with it matches the actual input string. In this case, the algorithm backtracks further.

The expansion rule declared by the macro is activated whenever the production has matched some section of the input. The code generated by the macro body is normalized, preprocessed, and tested again against productions of the original non-terminal. The matching rules for the generated code are the same as those for the original code; the generated code can also match a user-defined construct and, thus, transformed again. This can fail if the expansion rule itself generates code that is not well formed. In this case, backtracking proceeds further up (and might result in a syntax error).



## 16.6 Macro expansion code

The expansion rule is defined in the macro body. In the case of simple macros (define-as) it is given as parameterized code segment. The expansion segment is taken literally, except for occurrences of syntactic parameters. In the case of computed macros (define-as-computed) as an action block procedurally computes the expansion as a function of the syntactic parameters (available as string variables).

Syntactic parameters are formally declared in the match expression. Their actual values are the corresponding string part of the input text that was actually matched in each case (possibly empty strings).

Parameters can be of four kinds, as follows:

- a) Non-terminal parameters in the format `<[tag']non-terminal-type>` — referring to the corresponding non-terminal in the match expression. The reference is undefined if there is more than one non-terminal of the same type in the match expression without a unique tag.
- b) Non-terminal repetition parameters in the format: `<[tag']non-terminal-types>` (note the addition of the plural form 's' to the non-terminal type) — standing for to the entire submatch of non-terminal repetition in the match expression in the case of simple macros, or a list of string variable whose elements are the (zero or more) occurrences of the required non-terminal type in the case of computed macros.
- c) Labeled submatch parameters in the format `<caps-identifier>` (where *caps-identifier* is an identifier consisting only of capital letters and underscores)-referring to the submatch of the matched input labeled accordingly.
- d) Implicit submatch parameters in the format `<n>` (*n* being a positive integer) — referring to the *n*th submatch of the matched input. Submatches are the sections of input text matched inside grouping operators, option operators, or non-terminals. They are enumerated from left to right, the first of which is 1.

Two special replacement operators are available only in the scope of simple macros

- a) Default values for syntactic parameters in the format: `<param-selector/val>` (where *param-selector* is any of the above listed parameter kinds, and *val* being any text)-either referring to the corresponding syntactic parameter, or replaced with *val* when the actual value of the parameter is the empty string.
- b) A generated unique identifier in the format `<?>`, possibly appended to a legal identifier (e.g., `x<?>`). This is used for declaring entities (primarily variables) with names that are unique across all macro expansions, thereby avoiding the risk of name-collision upon expansion.

The code that is generated upon expansion is parsed again. It must be a legal construct of the same category as the one being replaced or a series thereof. Thus, macro expansion can be viewed as a local transformation on the syntax tree of a program, where some specific node is replaced by another without affecting its descendants or parent nodes.

### Example 1

This example is designed to demonstrate the principle of priority over productions determined by the order of definition (see [16.5.1](#)).

```
define <c_field1'struct_member> "C <num> <name>" as {
    %! <name>: uint(bits: <num>)
};

define <c_field2'struct_member> "C <exp> <name>" as {
    // The previous macro can never be matched because its pattern is
    // completely overshadowed by this one (<num> can always be parsed as
```

```

    // <exp>)
    %! <name>: uint(bits: <exp>)
};

define <c_field3'struct_member> "C <type'name> <name>" as {
    // This macro is not overshadowed by the previous (even though <name>
    // can be parsed as <exp>)
    %! <name>: <type'name>
};

extend sys {
    C byte f1;          // matched first by <c_field3'struct_member>
                        // replaced by '!% f1: byte'
    C 4*WORD_SIZE f2    // matched first by <c_field2'struct_member>
                        // replaced by '!% f2: uint(bits: 4*WORD_SIZE)'
}

```

### Example 2

The following is a definition of an action with the internal name `swap_var`. The match string contains two parsing element items, `<var1'exp>` and `<var2'exp>`, so the `<1>` in the third line corresponds to `<var1'exp>`, the first parsing element in the match string. The notation `<2>` could likewise be used for `<var2'exp>`. Thus, the third line could be written as `<1> = <2|z>`.

```

define <swap_var'action> "swap <var1'exp>[ <var2'exp>]" as {
    var tmp<?> := <var1'exp>;
    <1> = <var2'exp|z>;
    <var2'exp|z> = tmp<?>
};

extend sys {
    run() is also {
        var a := 5;
        var b := 9;
        var z := 13;

        swap a b;          // a becomes 9, b becomes 5
        print a, b, z;
        swap a;            // a becomes 13, z becomes 9
        print a, b, z
    }
}

```

### Example 3

This code illustrates the use of repetition operators within the match expression in a computed macro, and the corresponding use of list variables within its body.

```

define <multi_when'statement>
"extend \[<detr'name>,...\] <base'name> {<struct_member>;...}" as computed {
    for each in <detr'names> do {
        result = appendf("%s extend %s %s {%s}";
        result,it,<base'name>,str_join(<struct_members>,";"));
    };
};

// override the clock definition in a number of my_bfm's subtypes
extend [RED, BLUE, GREEN] my_bfm {

```

```
event clock is only rise('clock2')@sim;  
};
```



## 17. Print, checks, and error handling

The *e* language has many constructs that print an expression, check for errors in the DUT, or add exception handling and diagnostics to an *e* program.

### 17.1 print

<b>Purpose</b>	Print an expression	
<b>Category</b>	Action	
<b>Syntax</b>	<b>print</b> <i>exp</i> [...] [ <b>using</b> <i>print-options</i> ]	
<b>Parameters</b>	<i>exp</i>	Valid <i>e</i> expression.
	<i>print-options</i>	One or more of the print options, separated by commas ( , ). Print options are partly implementation-dependent. See also <a href="#">29.9</a> .

This prints the value of the given expression. Each type of expression has a default format in which it is printed.

- Scalars and string expressions print as *expression = value-of-expression*.
- Structs print in small tables, with a row for each field of a struct.

The predefined **do\_print()** method of a struct performs the printing. Changing the **do\_print()** method modifies the default format for the struct.

- a) Lists print in small tables with a row for each element in the list, except for scalar lists 16 bits or smaller, which print horizontally.
- b) Integers print leading zeros (0's), except for unbounded integers, 32-bit integers, or decimal integers.

Syntax example:

```
print byte_list using radix = hex
```

### 17.2 Handling DUT errors

There are several constructs that can be used to perform data or protocol checks on the DUT or to handle any errors that occur.

### 17.2.1 check that

<b>Purpose</b>	Perform a data comparison, and depending on the results, print a message	
<b>Category</b>	Action	
<b>Syntax</b>	<b>check</b> <i>[[name] that] bool-exp [else dut_error(message: string-exp, ...)]</i>	
<b>Parameters</b>	<i>name</i>	Any legal <i>e</i> identifier. When the argument <i>name</i> is used, the keyword <b>that</b> is mandatory.
	<i>bool-exp</i>	A Boolean expression that performs a data comparison.
	<i>message</i>	A string or an expression that can be converted to a string. If the <i>bool-exp</i> is FALSE, the message expressions are converted to strings, concatenated, and printed to the screen (and to the log file, if it is open).

This performs a data comparison, and depending on the results, prints a message. Use **check that** to track the number of failed checks with predefined **session** fields. When the **else dut\_error** clause is omitted, the *e* program uses the **check that** clause as the error message.

Syntax example:

```
check_count(i:int) is {
  check that i == expected_count else
    dut_error("Bad i: ", i)
}
```

#### 17.2.1.1 Using named check actions

The *name* argument is used to associate a name with a check action. Although check names have no direct application in *e* code, they provide a meaningful way for programmers and tools to identify the respective checks, for example, to determine their severity or collect coverage for them (such functionality is tool specific).

Rules:

- A named check is implicitly declared for a struct by associating a name with some check action in a method of that struct.
- Multiple check actions may be associated with the same name in the context of a given struct. All actions are taken to be implementing the very same logical check.
- A named check cannot be introduced in the scope of a struct if a check by the same name is already declared for a subtype of that struct.
- expect/assume struct members are checks too. They share the same namespace and so their *rule\_name* argument (if used) must not collide with a check *name*, nor vice versa.

Example:

```
extend bus_e {
  check() is also {
    check bus_cycle_length that num_of_cycles < 1000;
  }
}
```

### 17.2.2 dut\_error()

<b>Purpose</b>	Issue a DUT error message	
<b>Category</b>	Action	
<b>Syntax</b>	<b>dut_error</b> ( <i>message</i> : string-exp, ...) { <i>action</i> ; ...}	
<b>Parameters</b>	<i>message</i>	A string or an expression that can be converted to a string. The message expressions are converted to strings, concatenated, and printed to the screen (and to the log file, if it is open).
	<i>action</i>	A series of zero or more actions enclosed in braces ( { } ) and separated by semicolons ( ; ).

This issues a DUT error message. The action block, if it exists, is executed only when the error is actually issued (that is, if the check effect is not configured to IGNORE). This action is usually associated with an **if** action, a **check that** action, or an **expect** struct member. Calling **dut\_error()** directly is exactly equivalent to:

```
check that FALSE else dut_error()
```

NOTE—When **dut\_error()** is called directly or within an **expect**, **session.check\_ok** shall always be FALSE.

Syntax example:

```
if 'data_out' != 'data_in' then {
    dut_error("DATA MISMATCH: Expected ", 'data_in')
}
```

### 17.2.3 dut\_errorf()

<b>Purpose</b>	Issue a formatted DUT error message	
<b>Category</b>	Action	
<b>Syntax</b>	<b>dut_errorf</b> ( <i>format</i> : string, <i>item</i> : exp ...) { <i>action</i> ; ...}	
<b>Parameters</b>	<i>format</i>	A string expression containing a standard C formatting mask for each item (see <a href="#">29.7.3</a> )
	<i>item</i>	A legal <i>e</i> expression. String expressions shall be enclosed in double quotes (""). If the expression is a struct instance, the struct ID is printed. If the expression is a list, an error shall be issued.
	<i>action</i>	A series of zero or more actions enclosed in braces ( { } ) and separated by semicolons ( ; ).

This issues a formatted DUT error message. The action block, if it exists, is executed only when the error is actually issued (that is, if the check effect is not configured to IGNORE). This action is usually associated with an **if** action, a **check that** action, or an **expect** struct member.

### 17.2.4 dut\_error\_struct

<b>Purpose</b>	Define DUT error response
<b>Category</b>	Predefined struct
<b>Syntax</b>	<pre> <b>struct dut_error_struct {</b>   <b>get_message()</b>      : string;   <b>source_struct()</b>   : any_struct;   <b>source_location()</b> : string;   <b>source_struct_name()</b> : string;   <b>source_method_name()</b> : string;   <b>check_effect()</b>     : check_effect;   <b>set_check_effect(effect:check_effect);</b>   <b>write();</b>   <b>pre_error() is empty</b> <b>}</b> </pre>
<b>Struct members</b>	<b>get_message()</b> Returns the message defined by the temporal or data DUT check; this is printed by <b>dut_error_struct.write()</b> .
	<b>source_struct()</b> Returns a reference to the struct where the temporal or data DUT check is defined.
	<b>source_location()</b> Returns a string giving the line number and source module name, e.g., At line 13 in @checker.
	<b>source_struct_name()</b> Returns a string giving the name of the source struct, e.g., packet.
	<b>source_method_name()</b> Returns a string giving the name of the method containing the DUT data check, e.g., done ( ).
	<b>check_effect()</b> Returns the check effect of that DUT check, e.g., ERROR_AUTOMATIC.
	<b>set_check_effect()</b> Sets the check effect in this instance of the <b>dut_error_struct</b> . Call this method from <b>pre_error()</b> to change the check effect of selected checks.
	<b>pre_error()</b> The first method that is called when a DUT error occurs, unless the check effect is IGNORE. This method is defined as empty, unless extended by the user. Extending this method to modify error handling for a particular instance or set of instances of a DUT error.
	<b>write()</b> The method that is called after <b>dut_error_struct.pre_error()</b> is called when a DUT error happens. This method causes the DUT message to be displayed, unless the check effect is IGNORE. To perform additional actions, extend this method.

This defines the DUT error response. To modify the error response, extend either **write()** or **pre\_error()**. Only the **write()** and **pre\_error()** methods are called directly by *e* programs, but the other fields and predefined methods of **dut\_error\_struct** can also be used in extending **write()** or **pre\_error()**.

When a *dut\_error* is triggered, the runtime engine shall call **pre\_error()**, unless the **check\_effect** is set to IGNORE. Upon return of **pre\_error()**, the **write()** method is called, causing a message to be printed. Both these methods can be customized.

The other **dut\_error\_struct** methods previously listed (as *Struct members*) can also be used to create more meaningful error messages or the response can be conditioned based upon the **check\_effect**.



NOTE—Do not use **dut\_error\_struct.write()** to change the value of the check effect. Use **pre\_error()** instead.

Syntax example:

```

extend dut_error_struct {
    write() is also {
        if source_struct() is a XYZ_packet (p) then {
            print p.parity_calc()
        }
    }
}

```

### 17.2.5 set\_check()

<b>Purpose</b>	Set check severity
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>set_check</b> ( <i>static-match</i> : string, <i>check-effect</i> : keyword)
<b>Parameters</b>	<p><i>static-match</i>      A regular expression enclosed in double quotes (" "). Only checks whose message string matches this regular expression are modified. The pattern is matched against the constant part of the message string. The match string shall use the native <i>e</i> syntax or an AWK-like syntax (see <a href="#">4.11</a>). Any AWK-like syntax shall be enclosed in forward slashes, e.g., /<i>Vi</i>o/. Also, the * character in native <i>e</i> syntax matches only non-white characters. Use . . . to match white or non-white characters.</p>
	<p><i>check-effect</i>      <i>check-effect</i> is one of the following:</p> <ul style="list-style-type: none"> <li>a) <b>ERROR</b>—Issues an error message, increases <b>num_of_dut_errors</b>, breaks the run immediately and returns to the simulator prompt.</li> <li>b) <b>ERROR_BREAK_RUN</b>—Issues an error message, increases <b>num_of_dut_errors</b>, and breaks the run at the next cycle boundary.</li> <li>c) <b>ERROR_AUTOMATIC</b>—Issues an error message, increases <b>num_of_dut_errors</b>, breaks the run at the next cycle boundary, and performs the end of test checking and finalization of test data that is normally performed when <b>stop_run()</b> is called.</li> <li>d) <b>ERROR_CONTINUE</b>—Issues an error message, increases <b>num_of_dut_errors</b>, and continues execution.</li> <li>e) <b>WARNING</b>—Issues a warning, increases <b>num_of_dut_warnings</b>, and continues execution.</li> <li>f) <b>IGNORE</b>—Issues no messages, does not increase <b>num_of_dut_errors</b> or <b>num_of_dut_warnings</b>, and continues execution.</li> </ul>

This sets the severity or the check effect of specific DUT checks, so failing checks produce errors or warnings. This routine affects only checks that are currently loaded.

Syntax example:

```

extend sys {
    setup() is also {
        set_check("...", WARNING)
    }
}

```

## 17.3 Handling user errors

The *e* language has several constructs for handling user errors, such as file I/O errors or semantic errors. This subclause describes the constructs used for handling these kinds of errors.

- **warning()** issues a warning message when a given error occurs.
- **error()** issues an error message and exits when a given error is detected.
- **fatal()** issues an error message and exits to the OS prompt when a given error is detected.
- **try** defines an alternative response for fixing or bypassing an error.

Errors handled by these constructs do not increase the **session.num\_of\_dut\_errors** and **session.num\_of\_dut\_warnings** fields that are used to track DUT errors. In addition, the error responses defined with these constructs are not influenced by modifications to **dut\_error\_struct** or by **set\_check()** configurations.

See also the **run** option (29.9).

### 17.3.1 warning()

<b>Purpose</b>	Issue a warning message	
<b>Category</b>	Action	
<b>Syntax</b>	<b>warning</b> ( <i>message</i> : string-exp, ...)	
<b>Parameters</b>	<i>message</i>	A string or an expression that can be converted to a string. When the <b>warning</b> action is executed, the message expressions are converted to strings, concatenated, and printed to the screen.

This issues the specified warning error message. It does not affect execution.

Syntax example:

```
warning("len exceeds 50")
```

### 17.3.2 error()

<b>Purpose</b>	Issue an error message and halt all method execution	
<b>Category</b>	Action	
<b>Syntax</b>	<b>error</b> ( <i>message</i> : string-exp, ...)	
<b>Parameters</b>	<i>message</i>	A string or an expression that can be converted to a string. When the <b>error</b> action is executed, the message expressions are converted to strings, concatenated, and printed to the screen.

This issues the specified error message and halts all methods being currently run. The only exception to this is if the **error** action appears inside the first action block given in a **try** action. In that case, the *e* program jumps to the **else** action block within the **try** action and continues running. Calling **error()** directly is exactly equivalent to:

```
assert FALSE else error()
```

Unlike the **check that** action, **error()** does not use **dut\_error\_struct**.

Syntax example:

```
check_size() is {
    if pkt.size != LARGE then {
        error("packet size is ", pkt.size)
    }
}
```

### 17.3.3 fatal()

<b>Purpose</b>	Issue error message and exit to the OS prompt	
<b>Category</b>	Action	
<b>Syntax</b>	<b>fatal</b> ([ <i>status</i> : int,] <i>message</i> : string, ...)	
<b>Parameters</b>	<i>status</i>	A numeric value to be returned to the OS shell. If omitted, -1 is returned.
	<i>message</i>	A string or an expression that can be converted to a string. When the <b>fatal()</b> action is executed, the message expressions are converted to strings, concatenated, and printed to the screen.

Syntax example:

```
fatal(1, "Run-time error - exiting")
```

### 17.3.4 try

<b>Purpose</b>	Define an alternative response for fixing or bypassing an error	
<b>Category</b>	Action	
<b>Syntax</b>	<b>try</b> { <i>action</i> ; ...} [ <b>else</b> { <i>action</i> ; ...}]	
<b>Parameters</b>	<i>action</i>	A series of zero or more actions enclosed in braces ({ }) and separated by semicolons (;).

This executes the action block following **try**. If an error occurs, it executes the action block specified in the **else** clause. If no error occurs, the **else** clause is skipped. When the **else** clause is omitted, execution after an error continues normally from the first action following the **try** block.

Syntax example:

```
try {
    var my_file : file = files.open(file_name, "w", "Log file")
} else {
    warning("Could not open ", file_name,
        "; opening temporary log file sim.log")
}
```

## 17.4 Handling programming errors: **assert**

<b>Purpose</b>	Check the <i>e</i> code for correct behavior	
<b>Category</b>	Action	
<b>Syntax</b>	<b>assert</b> <i>bool-exp</i> [ <b>else error</b> ( <i>message</i> : string-exp, ...)]	
<b>Parameters</b>	<i>bool-exp</i>	A Boolean expression that checks the behavior of the code.
	<i>message</i>	A string or an expression that can be converted to a string. If the <i>bool-exp</i> is FALSE, the message expressions are converted to strings, concatenated, and printed to the screen (and to the log file, if it is open).

The *e* language has a special construct, the **assert** action, for handling certain programming errors, such as internal contradictions or invalid parameters. It checks the *e* code for correct behavior. Use this action to catch coding errors.

When an assert fails, it prints the specified error message, plus the line number and name of the file in which the error occurred. If the **else error** clause is omitted, **assert** prints a global error message.

When an error is encountered, **assert** stops the method being executed.

Syntax example:

```
assert a < 20
```

## 18. Methods

*e* methods are similar to C functions, Verilog tasks, and VHDL processes. An *e* method is an operational procedure containing actions that define its behavior. A method can have parameters, local variables, and a return value. A method can only be defined within a struct, and an instance of the struct needs to be created before executing the method. When a method is executed, it can manipulate the fields of that struct instance.

Methods can execute within a single point of simulation time (within zero time) or over multiple cycles. The first type of method is referred to as a *regular method*. The second type is called a *time-consuming method* or TCM.

TCMs can execute over multiple cycles and are used to synchronize processes in an *e* program with processes or events in the DUT. TCMs can contain actions that consume time, such as **wait**, **sync**, and **state machine**, and can call other TCMs. Within a single *e* program, multiple TCMs can execute in sequence or in parallel, along separate threads. A TCM can also have *internal branches*, which are multiple action blocks executing concurrently. See also [Clause 11](#), [Clause 12](#), [Clause 13](#), [Clause 14](#), and [32.1](#).

Methods defined in one module can later be overwritten, modified, or enhanced in subsequent modules by using the **extend** mechanism (see [18.1](#)).

*e* methods can be implemented in foreign languages such as C. Such methods shall be declared as **C routine**.

### 18.1 Rules for defining and extending methods

There are two phases in the definition of regular methods and TCMs: introduction and extension. A method needs to be introduced before it can be extended. The introduction can be in the same struct as the extension or in any struct from which this struct inherits, but it needs to precede the extension during file loading.

To introduce a method, use:

```
is [ C routine ]
is undefined | empty
```

To extend a method, use:

```
is (also | first | only)
is only C routine
```

**is** can also be used to extend a method in the following cases:

- The method was previously introduced with **is undefined** or **is empty** and has not been previously extended in this struct or in any struct that inherits from this struct.
- The method was previously introduced (and perhaps extended) in a struct from which this struct inherits, as long as the method has not already been extended in this struct or in any struct that inherits from this struct using **like**.

In these cases, using **is** after **is** or after **is (also | first | only)** in a **when** or **like** subtype is similar to using **is only** in this context, except an error message shall be generated if the method was already extended in this subtype or in any of its **like** subtypes (see [Clause 6](#)). The advantage of using **is** instead of **is only** is that an error shall be reported when the method extensions do not occur in the expected order.

[Table 37](#) summarizes the rules for introducing and extending methods. It uses the following keys:

- a) The *none* heading in the table indicates the method has not been introduced yet.
- b) The **only** heading represents **is also**, **is first**, and **is only**.
- c) The + character indicates “is allowed.”
- d) The – character indicates “is not allowed.”
- e) The C character indicates “is allowed” only in a **like** or **when** child, or one of its descendents.

**Table 37—Rules for method extension**

Extending by	Previous declaration				
	<i>none</i>	<b>undefined</b>	<b>empty</b>	<b>is</b>	<b>only</b>
<b>undefined</b>	+	–	–	–	–
<b>empty</b>	+	–	–	–	–
<b>is</b>	+	+	+	C	C
<b>only</b>	–	+	+	+	+

The following restrictions apply to all methods:

- Methods cannot be defined with variable argument lists; however, variable length list or structs with conditional fields can be passed in a list.

For example, the following method accepts a list of structs and performs appropriate operations on each struct in the list, depending on its type:

```
m(l: list of any_struct) is {
  for each (s) in l do {
    if s is a packet (p) then {};
    if s is a cell (c) then {}
  }
}
```

This method can then be called as follows:

```
m({my_cell; my_packet})
```

- Extending a method in a child (FALSE 'bye A) is allowed, even if the method has been extended in a sibling's descendant (stop bye A).
- When **is** is used after **is** or after **is (also | first | only)** in a **when** or **like** child or one of their descendents, then it cannot be used to redefine the method in the parent.
- Extending a child more than once with **is** shall generate an error.
- Extending a child with **is** after extending its descendant also shall generate an error.

#### Example

The following example shows how to use **is** to extend a method in a child after the method has already been introduced with **is** in the parent.

```
struct A {
  my_type() is {
    out("I am type A")
  }
};
```

```

struct B like A {};

struct C like B {
    my_type() is {
        out("I am type C, grandchild of A")
    }
}

```

The following subclauses describe the syntax for defining and extending methods. See also [6.3](#) and [6.7.2](#).

To understand how *e* code is serialized, see [Annex B](#).

### 18.1.1 method is [inline]

<b>Purpose</b>	Declare a regular method
<b>Category</b>	Struct member
<b>Syntax</b>	<i>method-name</i> ([ <i>parameter-list</i> ]) [: <i>return-type</i> ] <b>is</b> [ <b>inline</b> ] { <i>action</i> ;...}
<b>Parameters</b>	<i>method-name</i> A legal <i>e</i> name (see <a href="#">Clause 4</a> ).
	<i>parameter-list</i> A list composed of zero or more parameter declarations of the form <i>param-name</i> : [*] <i>param-type</i> [= <i>default-exp</i> ] separated by commas ( , ). a) <i>param-name</i> is a legal <i>e</i> name (see <a href="#">Clause 4</a> ). b) When an asterisk (*) is prefixed to a scalar parameter type, the parameter is passed by reference (see <a href="#">18.3</a> ). c) <i>param-type</i> —specifies the parameter type. d) <i>default-exp</i> , if given, must be a legal constant <i>e</i> expression of the right type (see <a href="#">18.3.4</a> ). The parentheses ( ( ) ) around the parameter list are required even if the parameter list is empty.
	<i>return-type</i> For methods that return values, this specifies the data type of the return value. See <a href="#">Clause 5</a> for more information.
	<i>action</i> ;...      A list of zero or more actions (see <a href="#">4.2.3</a> ). Actions that consume time are illegal in the action block of a regular method.

Defining a method as inline suggests to the *e* compiler to generate the code inline. The *e* compiler can place all the code for the method at each point in the code where the method is called. This inline expansion allows the compiler to optimize the inline method code for the best performance. Methods that are frequently called and involve computation, such as the following one, are good candidates for inline definition. This method takes two integers as arguments and returns an integer:

```

struct meth {
    get_free_area_size(size:int, taken:int):int is inline {
        result = size - taken
    }
}

```

In addition to the restrictions noted in [18.1](#), the following restrictions apply to inline methods:

- A method originally defined as **inline** cannot be redefined using **is only**, **is first**, or **is also**.

- Methods defined in **when** conditional struct members cannot be defined as inline.
- TCMs cannot be defined as inline.
- A breakpoint shall not be set on an inline method if there are compiled files that call the method.

See also [4.2.3](#) and [Clause 28](#).

Syntax example:

```
struct print {
    print_length(length:int) is {
        out("The length is: ", length)
    }
}
```

### 18.1.2 method @event is

<b>Purpose</b>	Declare a TCM
<b>Category</b>	Struct member
<b>Syntax</b>	<i>method-name</i> ([ <i>parameter-list</i> ]) [: <i>return-type</i> ] @ <i>event</i> <b>is</b> { <i>action</i> ;...}
<b>Parameters</b>	<i>method-name</i> A legal <i>e</i> name (see <a href="#">Clause 4</a> ).
	<i>parameter-list</i> A list composed of zero or more parameter declarations of the form <i>param-name</i> : [ <i>*</i> ] <i>param-type</i> [= <i>default-exp</i> ] separated by commas (, ). a) <i>param-name</i> is a legal <i>e</i> name (see <a href="#">Clause 4</a> ). b) When an asterisk (*) is prefixed to a scalar parameter type, the parameter is passed by reference (see <a href="#">18.3</a> ). c) <i>param-type</i> —specifies the parameter type. d) <i>default-exp</i> , if given, must be a legal constant <i>e</i> expression of the right type (see <a href="#">18.3.4</a> ). The parentheses ( ( ) ) around the parameter list are required even if the parameter list is empty.
	<i>return-type</i> For methods that return values, this specifies the data type of the return value. See <a href="#">Clause 5</a> for more information.
	@ <i>event</i> Specifies a default sampling event that determines the sampling points of the TCM. This event shall be a defined event in <i>e</i> and shall serve as the default sampling event for the TCM itself, as well as for any time-consuming actions, such as <b>wait</b> , within the TCM body. Other sampling points can also be added within the TCM. See also <a href="#">Clause 12</a> .
	<i>action</i> ;...      A list of zero or more actions (see <a href="#">4.2.3</a> ), either time-consuming actions or regular actions.

This defines a new TCM. TCMs shall implicitly synchronize on the designated sampling event upon entry, as if the action **sync** @*event* was added at the top of the TCM code (see [14.1.2](#)). No implicit synchronization occurs upon return.

Syntax example:

```
struct meth {
    main() @pclk is {
```



```

        wait @ready;
        wait [2];
        init_dut();
        emit init_complete
    }
}

```

### 18.1.3 method [**@event**] is (also | first | only | inline only)

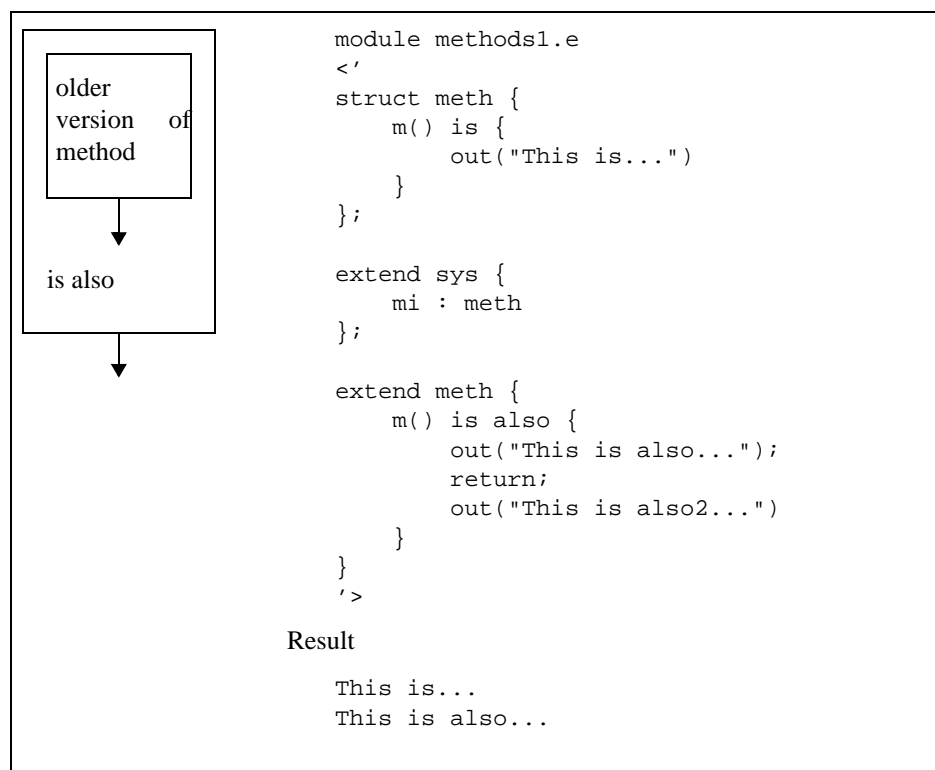
<b>Purpose</b>	Extend a regular method or a TCM
<b>Category</b>	Struct member
<b>Syntax</b>	<i>method-name</i> ([ <i>parameter-list</i> ]) [: <i>return-type</i> ] [ <b>@event-type</b> ] <b>is</b> ( <b>also</b>   <b>first</b>   <b>only</b>   <b>inline only</b> ) { <i>action</i> ;...}
<b>Parameters</b>	<i>method-name</i> The name of the original method.
	<i>parameter-list</i> Specifies the same parameter list defined in the original method; otherwise, a compile-time error shall be issued.
	<i>return-type</i> Specifies the same return value defined in the original method; otherwise, a compile-time error shall be issued.
	<b>@event</b> Specifies the same sampling event defined in the original method; otherwise, a compile-time error shall be issued.
	<b>also</b> The new definition refining the method implementation is called after the original body is executed.
	<b>first</b> The new definition refining the method implementation is called before the original body is executed.
	<b>only</b> The new definition refining the method implementation is called instead of the original body.
	<b>inline only</b> Replaces the original method definition with an inline definition. The original method shall be a regular method, not a TCM.
	<i>action</i> ;...        A list of zero or more actions (see 4.2.3). Time-consuming actions shall not be used in the action block of a regular method.

This replaces or extends the action block in the original method declaration with the specified action block. It carries the following restrictions:

- Methods that were originally defined as **inline** cannot be extended or redefined.
- An **is first** extension can manipulate the input of the original definition while an **is also** extension can make use of its output. When an **is first** extension changes the value of a parameter, the new value shall be passed to the existing definition, but if it assigns a value to **result**, this is not seen in the existing definition. The case with **is also** is the opposite; it is not affected if the original definition changes the value of the parameters, but in its scope, **result** holds the value returned from the existing definition.
- An **is also** extension is executed not only after the existing definition of the method for that same type, but also for all existing extensions in subtypes. The same applies for **is first**; it is executed first for objects of that type, regardless of the different existing definitions they might have.
- The following rules apply for **return** actions in extended methods, as illustrated in [Figure 9](#), [Figure 10](#), and [Figure 11](#).

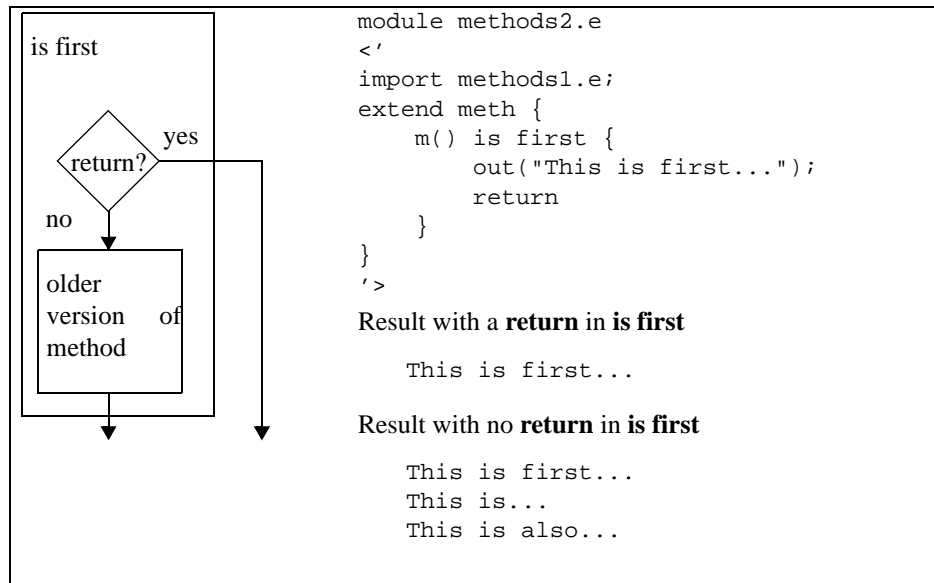
- 1) When an extension issues a **return**, any actions following that **return** within the extension itself are not executed.
- 2) When a method is extended with **is also**, the extension starts executing right after the older version of the method completes execution.
- 3) **is also** extensions are executed regardless of whether the older version of the method issues a **return** or not.
- 4) When a method is extended with **is first**, the older version of the method is never executed if the extension issues a **return**.
- 5) When a method is extended with **is only**, the older version of the method is never executed, whether the extension issues a **return** or not.

See also [31.3](#).



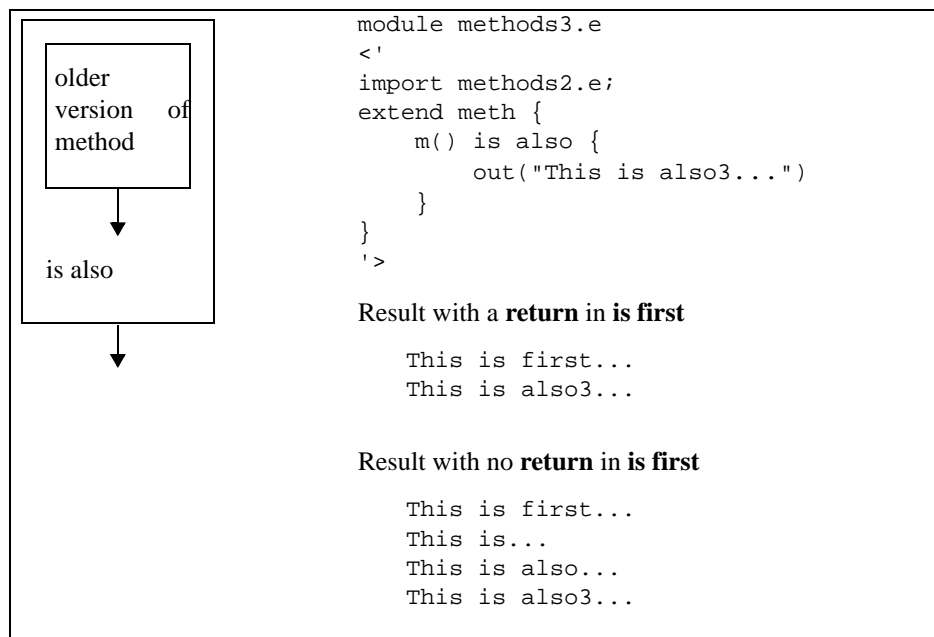
**Figure 9—Execution of is also method extension**

[Figure 9](#) shows how a method with an **is also** extension is executed. The older version executes first and then the **is also** extension. The “This is also2...” statement is not executed because it follows a **return**.

Figure 10—Execution of **is first** method extension

[Figure 10](#) shows the same method extended again, this time with **is first**. If a **return** statement is included in the **is first** extension, the older version of the method (the original method definition and the **is also** extension) do not execute. If the **return** statement is deleted, the **is first** extension executes and then the older version of the method executes.

[Figure 11](#) shows another extension with **is also**, which executes regardless of whether there is a **return** in the older version of the method or not.

Figure 11—Execution of **is first** method extension

Syntax example:

```
struct meth {
    run() is also {
        out("Starting main...");
        start main()
    }
}
```

#### 18.1.4 method [*@event*] is (undefined | empty)

<b>Purpose</b>	Declare an abstract method
<b>Category</b>	Struct member
<b>Syntax</b>	<i>method-name</i> ([ <i>parameter-list</i> ]) [ <i>: return-type</i> ] [ <i>@event-type</i> ] <b>is (undefined empty)</b>
<b>Parameters</b>	<i>method-name</i> A legal <i>e</i> name (see <a href="#">Clause 4</a> ).
	<i>parameter-list</i> A list composed of zero or more parameter declarations of the form <i>param-name</i> : [ <i>*</i> ] <i>param-type</i> separated by commas (,). a) <i>param-name</i> is a legal <i>e</i> name (see <a href="#">Clause 4</a> ). b)    When an asterisk (*) is prefixed to a scalar parameter type, the parameter is passed by reference (see <a href="#">18.3</a> ). c) <i>param-type</i> —specifies the parameter type. The parentheses ( ) around the parameter list are required even if the parameter list is empty.
	<i>return-type</i> For methods that return values, this specifies the data type of the return value. See <a href="#">Clause 5</a> for more information.
	<i>@event</i> Specifies a default sampling event that determines the sampling points of the TCM. This event shall be a defined event in <i>e</i> and shall serve as the default sampling event for the TCM itself, as well as for any time-consuming actions, such as <b>wait</b> , within the TCM body. Other sampling points can also be added within the TCM. See also <a href="#">Clause 12</a> .
	<b>undefined</b> No action block is defined for the method yet; an action block needs to be defined in a subsequent module before this method is called. A runtime error shall be issued if the action block is called before it is defined.
	<b>empty</b> The action block is empty, but no error is issued if it is called. Empty value-returning methods return the default value for the type.

This declares an abstract regular method or an abstract TCM with no defined functionality. *Abstract methods* are place holders that can be extended at a later point.

Syntax example:

```
struct packet {
    show() is undefined
}
```

## 18.2 Invoking methods

Before invoking a method, an instance of the struct that contains it needs to be created first. The call shall conform to the proper syntax and needs to be made from an appropriate context. See also [Clause 14](#) and [4.3](#).

- To invoke a TCM, use **tcm()**, **start tcm()**, or **compute** action (if the TCM returns a value).
- To call regular methods, use **method()** or **compute** action.
- To return from any method to the method that called it, use the **return** action (see [18.2.5](#)).

### 18.2.1 tcm()

<b>Purpose</b>	Call a TCM	
<b>Category</b>	Action or expression	
<b>Syntax</b>	[[ <i>struct-exp</i> ].] <i>method-name</i> ([ <i>parameter-list</i> ])	
<b>Parameters</b>	<i>struct-exp</i>	The pathname of the struct that contains the method. If the struct expression is missing, the implicit variable <b>it</b> is assumed. If both the struct expression and the period (.) are missing, the method name is resolved according to the scoping rules (see <a href="#">Clause 4</a> ).
	<i>method-name</i>	The method name as specified in the method definition.
	<i>parameter-list</i>	A list of zero or more expressions separated by commas (,), one expression for each parameter in the parameter list of the method declaration. Parameters are passed by their relative position in the list. Parameters at the end of the parameter list may be omitted if the method declaration specifies defaults for them (see <a href="#">18.3.4</a> ). The parentheses ( ) around the parameter list are required even if the parameter list is empty.

This calls a TCM. A TCM can only be called from another TCM. To invoke a TCM from within a regular method, use **start** (see [18.2.2](#)).

A TCM that does not return a value can be called or started (see [18.2.2](#)). A call of a TCM that does not return a value is syntactically an action. A call of a TCM that returns a value is an expression, and the return type of the TCM shall conform to the type of the variable or field to which it is assigned.

To call a value-returning method without using the value that is returned, use the **compute** action instead (see [18.2.4](#)).

A called TCM begins execution when its sampling event is emitted or immediately if the sampling event has already been emitted for the current tick. The calling TCM waits until the called TCM returns before continuing execution. In contrast, a started TCM runs in parallel with the TCM that started it.

Syntax example:

```
init_dut()
```

### 18.2.2 start tcm()

<b>Purpose</b>	Start a TCM	
<b>Category</b>	Action	
<b>Syntax</b>	<b>start</b> [[ <i>struct-exp</i> ].] <i>method-name</i> ([ <i>parameter-list</i> ])	
<b>Parameters</b>	<i>struct-exp</i>	The pathname of the struct that contains the method. If the struct expression is missing, the implicit variable <b>it</b> is assumed. If both the struct expression and the period (.) are missing, the method name is resolved according to the scoping rules (see <a href="#">Clause 4</a> ).
	<i>method-name</i>	The method name as specified in the method definition.
	<i>parameter-list</i>	A list of zero or more expressions separated by commas (,), one expression for each parameter in the parameter list of the method declaration. Parameters are passed by their relative position in the list. Parameters at the end of the parameter list may be omitted if the method declaration specifies defaults for them (see <a href="#">18.3.4</a> ). The parentheses ( ) around the parameter list are required even if the parameter list is empty.

This starts a TCM. It can be used within another method, either a TCM or a regular method. A TCM that has a return value cannot be started with a **start** action.

A started TCM can be scheduled for execution only once the starting TCM has been suspended (see [14.1.1](#)). A started TCM runs in parallel with the method that started it.

NOTE—The recommended way to start an initial TCM, which can then invoke other TCMs, is to extend the related struct's predefined **run()** method (see [28.2.2.4](#)).

Syntax example:

```
start main()
```

### 18.2.3 method()

<b>Purpose</b>	Call a regular method	
<b>Category</b>	Action or expression	
<b>Syntax</b>	[[ <i>struct-exp</i> ].] <i>method-name</i> ([ <i>parameter-list</i> ])	
<b>Parameters</b>	<i>struct-exp</i>	The pathname of the struct that contains the method. If the struct expression is missing, the implicit variable <b>it</b> is assumed. If both the struct expression and the period (.) are missing, the method name is resolved according to the scoping rules (see <a href="#">Clause 4</a> ).
	<i>method-name</i>	The method name as specified in the method definition.
	<i>parameter-list</i>	A list of zero or more expressions separated by commas (,), one expression for each parameter in the parameter list of the method declaration. Parameters are passed by their relative position in the list. Parameters at the end of the parameter list may be omitted if the method declaration specifies defaults for them (see <a href="#">18.3.4</a> ). The parentheses ( ) around the parameter list are required even if the parameter list is empty.

This calls a regular method. A call of a method that does not return a value is syntactically an action. A call of a method that returns a value is an expression, and the return type of the method shall conform to the type of the variable or the field to which it is assigned.

To call a value-returning method without using the value that is returned, use the **compute** action instead (see [18.2.4](#)).

Syntax example:

```
tmp1 = get_free_area_size(size, taken)
```

### 18.2.4 compute method() or tcm()

<b>Purpose</b>	Compute a regular method or TCM	
<b>Category</b>	Action	
<b>Syntax</b>	<b>compute</b> [[ <i>struct-exp</i> ].] <i>method-name</i> ([ <i>parameter-list</i> ])	
<b>Parameters</b>	<i>struct-exp</i>	The pathname of the struct that contains the method. If the struct expression is missing, the implicit variable <b>it</b> is assumed. If both the struct expression and the period (.) are missing, the method name is resolved according to the scoping rules (see <a href="#">Clause 4</a> ).
	<i>method-name</i>	The method name as specified in the method definition.
	<i>parameter-list</i>	A list of zero or more expressions separated by commas (,), one expression for each parameter in the parameter list of the method declaration. Parameters are passed by their relative position in the list. Parameters at the end of the parameter list may be omitted if the method declaration specifies defaults for them (see <a href="#">18.3.4</a> ). The parentheses ( ) around the parameter list are required even if the parameter list is empty.

This calls a value-returning method or TCM without using the value that is returned.

Syntax example:

```
if 'top.b' > 15 then {
    compute inc_counter()
}
```

### 18.2.5 return

<b>Purpose</b>	Return from regular method or a TCM	
<b>Category</b>	Action	
<b>Syntax</b>	<b>return</b> [ <i>exp</i> ]	
<b>Parameters</b>	<i>exp</i>	In value-returning methods, an expression specifying the return value is required in each <b>return</b> action. In non-value-returning methods, expressions are not allowed in <b>return</b> actions.

This returns immediately from the current method to the method that called it. The execution of the calling method then continues.

It is not always necessary to provide a **return** action. When a value returning method ends without a **return** action, the value of **result** is returned.

- The **return** action needs to be used carefully in method extensions (see [18.1.3](#)).
- Any actions that follow a **return** action in the method definition or a method extension are ignored.

NOTE—For value-returning methods, instead of using **return**, the special variable **result** can be assigned and its value shall be returned instead (see [4.3.3.3](#)).

Syntax example:

```
return i*i
```

## 18.3 Parameter passing

How a parameter is passed depends on whether the parameter is scalar or compound.

### 18.3.1 Scalar parameter passing

Scalar parameters include numeric, Boolean, and enumerated types. When a scalar parameter is passed to a method, by default, the value of the parameter is passed. This is called *passing by value*. Any change to the value of that parameter by the method applies only within that method instance and is lost when the method returns.

To allow a method to modify the parameter, prefix the parameter type with an asterisk (\*). This is called *passing by reference*. The asterisk shall only be used in the method definition, not in the method call. See also [18.3.3](#).



*Example 1*

The `increment()` method defined as follows increments the value of the parameter passed to it.

```
increment (cnt:int) is {
    cnt = cnt + 1
};

m() is {
    var tmp : int = 7;

    increment(tmp);
    print tmp
}
```

However, since `cnt` is passed by value, the variable `tmp` retains its original value, and the print statement displays:

```
tmp = 7
```

*Example 2*

If `increment()` is changed so the parameter can be modified [prefixing with an asterisk (\*)]:

```
increment (cnt:*int) is {
    cnt = cnt + 1
};

m() is {
    var tmp : int = 7;

    increment(tmp);
    print tmp
}
```

then the print statement displays:

```
tmp = 8
```

**18.3.2 Compound parameter passing**

Compound parameters are either structs or lists. Passing a struct or a list to a method allows the method to modify the struct fields and the list items. This is called *passing by reference*. To completely replace the struct or list, prefix the struct type or list with an asterisk (\*). See also [18.3.3](#).

*Example 1*

The `increment_list()` method defined as follows accepts a list.

```
increment_list(cnt:list of int) is {
    for each in cnt do {
        cnt[index] += 1
    }
}
```

If a list of integers is passed to it, each item in the list reflects its incremented value after the method returns.

*Example 2*

The `create_if_illegal()` method defined as follows accepts a struct instance of type `packet`.

```
create_if_illegal(pkt:*packet) is {
    if not pkt.legal then {
        pkt = new
    }
}
```

If it determines that the `legal` field of struct instance is `FALSE`, it allocates a new struct instance of type `packet`.

### 18.3.3 Passing by reference

Several restrictions apply when passing parameters *by reference*, as follows:

- There is no automatic casting to a reference parameter. Thus, an attempt to pass a variable that is a different type than the type declared within the calling method shall result in a compile-time error.
- A list element shall not be passed by reference.
- An expression that cannot be placed on the LHS of an assignment shall not be passed by reference.
- Called TCMs can accept reference parameters, but started TCMs shall not.

### 18.3.4 Default parameter values

Default values may be defined for method or TCM parameters using the assignment (`=`) operator. The following rules hold of parameter defaults:

- If a default is specified for some parameter, providing a value to that parameter in the invocation is optional. If a value is provided it is passed to the respective parameter, and otherwise the default is passed.
- Defaults may be specified in the method declaration only for consecutive parameters at the end of the parameter list. A default may not be specified to a parameter that is followed by a parameter without default.
- Actual parameters in the invocation are associated with formal parameters of the method in order from left to right. Parameters at the end of the parameter list may be omitted in the call if defaults exist for them. There is no way in a method call to pass a value to a parameter without providing values to all parameters preceding it.
- Default values may be specified only in the initial declaration of the method. No default value specification is allowed in method extensions.
- Default values are specified with constant expressions only. No variables of any kind are allowed within these expressions.

## 18.4 Using the C interface

This syntax allows the user to attach a C implementation to an *e* method or routine header, as well as exporting *e* types to C.

### 18.4.1 routine ... is C routine

<b>Purpose</b>	Declare a global <i>e</i> routine that is implemented in C
<b>Category</b>	Statement
<b>Syntax</b>	<b>routine</b> <i>e-routine-name</i> ([ <i>parameter-list</i> ]) [ <i>:result-type</i> ] <b>is C routine</b> [ <i>c-routine-name</i> ]
<b>Parameters</b>	<i>e-routine-name</i> The name used to call the C routine from <i>e</i> .
	<i>parameter-list</i> A list composed of zero or more parameter declarations of the form <i>param-name</i> : [*] <i>param-type</i> separated by commas ( , ). <ul style="list-style-type: none"> <li>a) <i>param-name</i> is a legal <i>e</i> name (see <a href="#">Clause 4</a>).</li> <li>b) When an asterisk (*) is prefixed to a scalar parameter type, the parameter is passed by reference (see <a href="#">18.3</a>).</li> <li>c) <i>param-type</i>—specifies the parameter type.</li> </ul> The parentheses ( ( ) ) around the parameter list are required even if the parameter list is empty.
	<i>result-type</i> The type of value returned by the C routine.
	<i>c-routine-name</i> The name of the routine as defined in C.

Declares a global *e* routine with the specified name, parameters, and result-type. The global *e* routine is implemented by the specified C routine. If the optional *c-routine-name* is omitted, the assumed name of the C routine is the *e-routine-name*. A C routine cannot have the same name as an existing method of **global**.

Syntax example:

```
routine mean_sqrt(l:list of int): int is C routine wrap_mean_sqrt
```

### 18.4.2 method ... is C routine

<b>Purpose</b>	Declare an <i>e</i> method that is implemented in C
<b>Category</b>	Struct member
<b>Syntax</b>	<i>e-method-name</i> ([ <i>parameter-list</i> ]) [ <i>:result-type</i> ] <b>is [only] C routine</b> [ <i>c-routine-name</i> ]
<b>Parameters</b>	<i>e-method-name</i> The name used to call the C routine from <i>e</i> .
	<i>parameter-list</i> A list composed of zero or more parameter declarations of the form <i>param-name</i> : [ <i>*</i> ] <i>param-type</i> separated by commas ( , ). a) <i>param-name</i> is a legal <i>e</i> name (see <a href="#">Clause 4</a> ). b)    When an asterisk (*) is prefixed to a scalar parameter type, the parameter is passed by reference (see <a href="#">18.3</a> ). c) <i>param-type</i> —specifies the parameter type. The parentheses ( ( ) ) around the parameter list are required even if the parameter list is empty.
	<i>result-type</i> The type of value returned by the C routine.
	<i>c-routine-name</i> The name of the routine as defined in C.

Declares an *e* method that is implemented by a C routine. When the *e* method is called, the C routine is executed. If the declaration states **is only**, the C implementation is called instead of the original body. When an *e* method whose functionality is implemented as a C routine is called, the struct instance of the called method is passed as the first parameter to the C routine.

Syntax example:

```
encrypt(data:byte): byte is C routine proprietary_encrypt
```

### 18.4.3 C export

<b>Purpose</b>	Export the <i>e</i> declared type or method to C
<b>Category</b>	Statement
<b>Syntax</b>	<b>C export</b> <i>type-name</i> <b>C export</b> <i>type-name.method-name()</i>
<b>Parameters</b>	<i>type-name</i> The type to export.
	<i>method-name</i> The method to export.

Marks the *e* declared type or method for inclusion in a generated C header file. To use *e* data types and *e* methods in C files, first create a C header file that declares these types and methods, and then include it in the C file. The following considerations also apply:

- The **export** statement can appear in any *e* module (where the types or methods are known).
- **when** subtypes (for example, small packet) cannot be exported.

Syntax example:

```
C export packet;  
C export packet.add()
```

## 19. Creating and modifying *e* variables

This clause describes how to create and assign values to *e* variables.

### 19.1 About *e* variables

An *e* variable is a named data object of a declared type. *e* variables are declared and manipulated in methods. They are dynamic; they do not retain their values across subsequent calls to the same method. Some *e* actions create implicit variables (see [4.3.3](#)).

The scope of an *e* variable is the action block that encloses it. If a method contains nested action blocks, variables in the inner scopes hide the variables in the outer scopes. Variable scoping is described in more detail in [4.3](#).

The following subclauses describe the actions that create and modify *e* variables explicitly.

### 19.2 var

<b>Purpose</b>	Variable declaration	
<b>Category</b>	Action	
<b>Syntax</b>	<b>var</b> <i>name</i> [: [ <i>type</i> ] [= <i>exp</i> ]]	
<b>Parameters</b>	<i>name</i>	A legal <i>e</i> name.
	<i>type</i>	A declared <i>e</i> type. The type can be omitted if the variable name is the same as the name of a struct type or if the variable is assigned a typed expression.
	<i>exp</i>	The initial value of the variable. If no initial value is specified, the variables are initialized to 0 for integer types, NULL for structs, FALSE for Boolean types, and empty lists for lists.

This declares a new variable with the specified name as an element or list of elements of the specified type and having an optional initial value. The **var** action is legal in any place that an action is legal and the variable is recognized from that point on. The scope of an *e* variable starts at the **var** action and ends at the bottom of the action block that encloses it. If a method contains nested action blocks, variables in the inner scopes hide the variables in the outer scopes.

Type information for **var** actions without an explicit type shall be inferred according to the following rules:

- If the *name* is the same as a declared type, that *type* shall be used.
- If the initial value expression is typed, that *type* shall be used.
- Otherwise, an error shall occur.

See also [4.3](#).

Syntax example:

```
var a : int
```

**19.3 =**

<b>Purpose</b>	Simple assignment	
<b>Category</b>	Action	
<b>Syntax</b>	<i>lhs-exp</i> <b>=</b> <i>exp</i>	
<b>Parameters</b>	<i>lhs-exp</i>	A legal <i>e</i> expression that evaluates to a variable of a method, a global variable, a field of a struct, or an HDL object. The expression can contain the list index operator [ <i>n</i> ], the bit access operator [ <i>i</i> : <i>j</i> ], or the bit concatenation operator % { }.
	<i>exp</i>	A legal <i>e</i> expression, either an untyped expression (such as an HDL object) or an expression of the same type as the <i>lhs-exp</i> .

This assigns the value of the RHS expression to the LHS expression (see also [Clause 5](#)).

NOTE—There are two other places within the *e* language that make use of the equal sign. These are a double equal sign (==) for specifying equality in Boolean expression and a triple equal sign (===) for the Verilog-like identity operator. Do not confuse these two operators with the single equal sign (=) assignment operator.

Syntax example:

```
sys.u = 0x2345
```

**19.4 op=**

<b>Purpose</b>	Compound assignment	
<b>Category</b>	Action	
<b>Syntax</b>	<i>lhs-exp</i> <i>op</i> <b>=</b> <i>exp</i>	
<b>Parameters</b>	<i>lhs-exp</i>	A legal <i>e</i> expression that evaluates to a variable of a method, a global variable, a field of a struct, or an HDL object.
	<i>op</i>	A binary operator, including binary bitwise operators (except ~), the Boolean operators <b>and</b> and <b>or</b> , and the binary arithmetic operators.
	<i>exp</i>	A legal <i>e</i> expression of the same type as the <i>lhs-exp</i> .

This performs the specified operation on the two expressions and assigns the result to the LHS expression.

Syntax example:

```
sys.c.count1 += 5
```

## 19.5 <=

<b>Purpose</b>	Delayed assignment
<b>Category</b>	Action
<b>Syntax</b>	<code>[<i>struct-exp</i>.]<i>field-name</i> &lt;= <i>exp</i></code>
<b>Parameters</b>	<i>struct-exp</i> A legal <i>e</i> expression that evaluates to a struct. The default is <b>me</b> .
	<i>field-name</i> A field of the struct referenced by <i>struct-exp</i> .
	<i>exp</i> A legal <i>e</i> expression, either an untyped expression (such as an HDL object) or an expression of the same type as the <i>lhs-exp</i> .

The delayed assignment action assigns a struct field just before the next **@sys.new\_time** (see [11.4.2](#)) after the action. The purpose is to support raceless coding in *e* by providing the same results regardless of the evaluation order of TCMs and TEs. Both expressions are evaluated immediately (not delayed) in the current context. The assignment is not considered a time-consuming action; it can be used in both TCMs and in regular methods, in **on** action blocks and in **exec** action blocks.

- If a field has multiple delayed assignments in the same cycle, they are performed in the specified order. The final result is taken from the last delayed assignment action.
- Unlike in HDL languages, the delayed assignment in *e* does not emit any events; thus, zero-delay iterations are not supported.
- The LHS expression in the delayed assignment action can only be a field. Unlike the assignment action, the delayed assignment action does not accept any assignments to the following:
  - Variable of a method
  - List item
  - Bit
  - Bit slice
  - Bit concatenation expression

### Example

The following example shows how delayed assignment provides raceless coding. In this example, there is one `incrementing()` TCM, which repeatedly increments the `sys.a` and `sys.da` fields, and one `observer()` TCM, which observes their value.

```
<'
extend sys {
  !a : int;
  !da : int;

  incrementing()@any is {
    for i from 1 to 5 do {
      a = a + 1;
      da <= da + 1;
      wait cycle
    };
    stop_run()
  };

  observer()@any is {
```

```

        while TRUE do {
            out("observing 'a' as ", a, " observing 'da' as ", da);
            wait cycle
        }
    };

    run() is also {
        start observer();
        start incrementing()
    }
}
'>

```

The following results show the value of `sys.a` observed by the `observer()` TCM is order-dependent, depending whether `observer()` is executed before or after `incrementing()`. The observed value of `sys.da`, however, is independent of the execution order. Even if `incrementing()` runs first, `sys.da` gets its incremented value just before the next **new\_time** event, and thus is not be seen by `observer()`.

If `observer()` runs before `incrementing()`:

```

observing 'a' as 0 observing 'da' as 0
-----
observing 'a' as 1 observing 'da' as 1
-----
observing 'a' as 2 observing 'da' as 2
-----
observing 'a' as 3 observing 'da' as 3
-----
observing 'a' as 4 observing 'da' as 4
-----

```

If `incrementing()` runs before `observer()`:

```

observing 'a' as 1 observing 'da' as 0
-----
observing 'a' as 2 observing 'da' as 1
-----
observing 'a' as 3 observing 'da' as 2
-----
observing 'a' as 4 observing 'da' as 3
-----
observing 'a' as 5 observing 'da' as 4
-----

```



## 20. Packing and unpacking

Packing and unpacking operate on scalars, strings, lists, and structs. The following subclauses show how to perform basic packing and unpacking of these data types using two of the *e* basic packing tools, the **pack()** and **unpack()** methods. See also [5.2](#).

### 20.1 Basic packing

The following subclauses detail how to pack, unpack, and swap data.

#### 20.1.1 pack()

<b>Purpose</b>	Perform concatenation and type conversion	
<b>Category</b>	Pseudo-routine	
<b>Syntax</b>	<b>pack</b> ( <i>option</i> :pack option, <i>item</i> : exp, ...): list of bit	
<b>Parameters</b>	<i>option</i>	<p>For basic packing, this parameter is one of the following choices. See <a href="#">20.3</a> for information on other pack options.</p> <ul style="list-style-type: none"> <li>a) <b>packing.high</b>—Places the LSB of the last physical field declared or the highest list item at index [0] in the resulting list of bit. The MSB of the first physical field or lowest list item is placed at the highest index in the resulting list of bit.</li> <li>b) <b>packing.low</b>—Places the LSB of the first physical field declared or the lowest list item at index [0] in the resulting list of bit. The MSB of the last physical field or highest list item is placed at the highest index in the resulting list of bit.</li> <li>c) <b>NULL</b>—If NULL is specified, the global default is used. This global default is set initially to <b>packing.low</b>.</li> </ul>
	<i>item</i>	A legal <i>e</i> expression that is a path to a scalar or a compound data item, such as a struct, field, list, or variable.

This performs concatenation of items, including items in a list or fields in a struct, in the order specified by the pack options parameter and returns a list of bits. This method also performs type conversion between any of the following:

- Scalars
- Strings
- Lists and list subtypes (derived structs)

Packing is commonly used to prepare high-level *e* data into a form that can be applied to a DUT. It operates on scalar or compound (struct or list) data items. Pack expressions are untyped expressions. In many cases, the *e* program can deduce the required type from the context of the pack expression (see [5.2](#)). An unbounded integer cannot be packed.

Syntax example:

```
i_stream = pack(packing.high, opcode, operand1, operand2)
```

### 20.1.2 unpack()

<b>Purpose</b>	Unpack a bit stream into one or more expressions	
<b>Category</b>	Pseudo-routine	
<b>Syntax</b>	<b>unpack</b> ( <i>option</i> : pack option, <i>value</i> : exp, <i>target1</i> : exp [, <i>target2</i> : exp, ...])	
<b>Parameters</b>	<i>option</i>	For basic packing, this parameter is one of the following choices. See <a href="#">20.3</a> for information on other pack options. <ul style="list-style-type: none"> <li>a) <b>packing.high</b>—Places the MSB of the list of bits at the MSB of the first field or lowest list item. The LSB of the list of bits is placed into the LSB of the last field or highest list item.</li> <li>b) <b>packing.low</b>—Places the LSB of the list of bits at the LSB of the first field or lowest list item. The MSB of the list of bits is placed into the MSB of the last field or highest list item.</li> <li>c) <b>NULL</b>—If NULL is specified, the global default is used. This global default is set initially to <b>packing.low</b>.</li> </ul>
	<i>value</i>	A scalar expression or list of scalars that provides a value that is to be unpacked.
	<i>target1, target2</i>	One or more expressions separated by commas ( , ). Each expression is a path to a scalar or a compound data item, such as a struct, field, list, or variable.

This converts a raw bit stream into high-level data by storing the bits of the *value* expression into the *target* expressions. If the value expression is not a list of bit, it is first converted (see [20.5](#)) into a list of bit by calling **pack()** using **packing.low**. Then, the list of bits is unpacked into the target expressions.

The *value* expression is allowed to have more bits than are consumed by the *target* expressions. In that case, if **packing.low** is used, the extra high-order bits are ignored; if **packing.high** is used, the extra low-order bits are ignored.

Unpacking is commonly used to convert raw bit stream output from the DUT into high-level *e* data. It operates on scalar or compound (struct or list) data items.

Syntax example:

```
unpack(packing.high, lob, s1, s2)
```

### 20.1.3 swap()

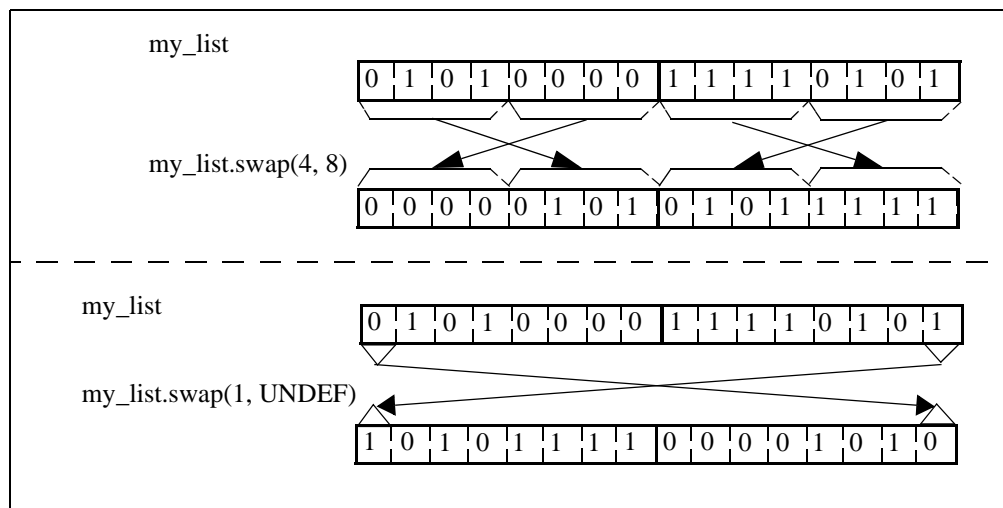
<b>Purpose</b>	Swap small bit chunks within larger chunks	
<b>Category</b>	Pseudo-routine	
<b>Syntax</b>	<i>list-of-bit</i> . <b>swap</b> ( <i>small</i> : int, <i>large</i> : int): list of bit	
<b>Parameters</b>	<i>small</i>	An integer that is a factor of <i>large</i> .
	<i>large</i>	An integer that is either UNDEF or a factor of the number of bits in the entire list. If UNDEF, the method reverses the order of small chunks within the entire list, e.g., <code>lob.swap(1, UNDEF)</code> is the same as <code>lob.reverse()</code> .

This predefined list method accepts a list of bits, changes the order of the bits, and then returns the reordered list of bits. This method is often used in conjunction with **pack()** or **unpack()** to reorder the bits in a bit stream going to or coming from the DUT.

- If *large* is not a factor of the number of bits in the entire list, an error message shall result.
- If *small* is not a factor of *large*, an error message shall also result. The only exception is if *large* is UNDEF and *small* is not a factor, the swap is not performed and no error message is issued.

#### Example

[Figure 12](#) shows two swaps. The first swap reverses the order of nibbles in every byte. The second swap reverses the whole list.



**Figure 12—Swapping examples**

Syntax example:

```
s2 = s1.swap(2, 4)
```

## 20.2 Predefined pack options

This subclause details the predefined pack options: **packing.high**, **packing.low**, and **packing.global\_default**.

### 20.2.1 pack\_options struct

The predefined instances are all instances of the **pack\_options** struct. The **pack\_options** declaration is:

```
struct pack_options {
    reverse_fields      : bool;
    reverse_list_items : bool;
    final_reorder       : list of int;
    scalar_reorder      : list of int
}
```

The following subclauses describe each of its fields.

#### 20.2.1.1 reverse\_fields

If this flag is set to **FALSE**, the fields in a struct are packed in the order they appear in the struct declaration; if **TRUE**, they are packed in reverse order. The default is **FALSE**.

#### 20.2.1.2 reverse\_list\_items

If this flag is set to **FALSE**, the items in a list are packed in ascending order; if **TRUE**, they are packed in descending order. The default is **FALSE**.

#### 20.2.1.3 final\_reorder

The **final\_reorder** field can be used after packing each element in the packing expression to perform final swapping on the resulting bit stream. The list in the **final\_reorder** field shall include an even number of items. Each pair of items in the list is the parameter list of a **swap()** operation (see [20.1.3](#)). To perform multiple swaps, use multiple pairs of parameters (each parameter pair corresponds to a pair of swap parameters).

#### 20.2.1.4 scalar\_reorder

The **scalar\_reorder** field can be used to perform one or more **swap()** operations on each scalar before packing. The list in the **scalar\_reorder** field shall include an even number of items. Each pair of items in the list is the parameter list of a **swap()** operation (see [20.1.3](#)). To perform multiple swaps, use multiple pairs of parameters (each parameter pair corresponds to a pair of swap parameters).

### 20.2.2 Predefined settings

[Table 38](#) shows the corresponding settings for each of the predefined pack options (**packing.high**, **packing.low**, and **packing.global\_default**). To customize a packing option, see [20.3](#).

**Table 38—Predefined packing options**

Flag	packing.high	packing.low	packing.global_default
<b>reverse_fields</b>	True	False	False
<b>reverse_list_items</b>	True	False	False
<b>final_reorder</b>	0	0	0
<b>scalar_reorder</b>	0	0	0

### 20.2.3 packing.high

This **pack\_options** instance traverses the source fields or variables in the reverse order from the order in which they appear in code, placing the LSB of the last field or list item at index [0] in the resulting list of bit. The MSB of the first field or list item is placed at the highest index in the resulting list of bit.

### 20.2.4 packing.low

This **pack\_options** instance traverses the source fields or variables in the order they appear in code, placing the LSB of the first field or list item at index [0] in the resulting list of bit. The MSB of the last field or list item is placed at the highest index in the resulting list of bit.

### 20.2.5 packing.global\_default

This **pack\_options** instance is used when the first parameter of **pack()**, **unpack()**, **do\_pack()**, or **do\_unpack()** is NULL. It has the same flags as **packing.low**.

## 20.3 Customizing pack options

Each of the predefined instances defined within the **pack\_options** struct (see [20.2.1](#)) can also be modified. To customize the packing options:

- Create an instance of the **pack\_options** struct;
- Modify one or more of its fields;
- Pass the struct instance as the first parameter to **pack()**, **unpack()**, **do\_pack()**, or **do\_unpack()**.

## 20.4 Packing and unpacking specific types

This subclause defines how to operate on structs, lists, scalars, and strings.

### 20.4.1 Packing and unpacking structs

Packing a struct creates an ordered bit stream from all the physical fields (marked with %) in the struct, starting with the first physical field declared. Other fields (called *virtual fields*) are ignored by the packing process. If a physical field is of a compound type (struct or list), the packing process descends recursively into the struct or list.

Unpacking a bit stream into a struct fills the physical fields of the struct, starting with the first field declared and proceeding recursively through all the physical fields of the struct. Unpacking a bit stream into a field that is a list follows some additional rules (see [20.4.2](#)).

Unpacking a struct that has not yet been allocated (with **new**) causes the *e* program to allocate the struct and run the struct's **init()** method. Unlike **new**, the struct's **run()** method is not called.

#### 20.4.1.1 Customizing packing for a particular struct

A struct is packed or unpacked using its predefined methods **do\_pack()** and **do\_unpack()**. It is possible to modify these predefined methods for a particular struct. These methods are called automatically whenever data is packed from or unpacked into the struct.

##### 20.4.1.1.1 **do\_pack()**

<b>Purpose</b>	Pack the physical fields of the struct
<b>Category</b>	Predefined method of <b>any_struct</b>
<b>Syntax</b>	<b>do_pack</b> ( <i>options</i> :pack options, <i>l</i> : *list of bit)
<b>Parameters</b>	<i>options</i> This parameter is an instance of the <b>pack_options</b> struct (see <a href="#">20.3</a> ).
	<i>l</i> An empty list of bits that is extended as necessary to hold the data from the struct fields.

The **do\_pack()** method of a struct is called automatically whenever the struct is packed. This method appends data from the physical fields (the fields marked with %) of the struct into a list of bits according to flags determined by the pack options parameter. The virtual fields of the struct are skipped. The method shall issue a runtime error message if this struct has no physical fields.

The **do\_pack()** method can be extended to create a unique packing scenario for that struct by creating a custom **pack\_options** instance (see [20.3](#)).

The following considerations also apply:

- Do not call the **do\_pack()** method of any struct directly, e.g., `my_struct.do_pack()`. Use **pack()** instead, e.g., `pack(packing.high, my_struct)`.
- Do not call **pack(me)** in the **do\_pack()** method. This causes infinite recursion. Call **packing.pack\_struct(me)** instead.
- Append the results of any pack operation within **do\_pack()** to the empty list of bits referenced in the **do\_pack()** parameter list.
- If the **do\_pack()** method is modified and then physical fields are added later in an extension to the struct, that **do\_pack()** might need to be modified again.

#### Example

The following assignment to `lob`

```
lob = pack(packing.high, i_struct, p_struct)
```

makes the following calls to the **do\_pack** method of each struct, where `tmp` is an empty list of bits:

```
i_struct.do_pack(packing.high, *tmp);
```

```
p_struct.do_pack(packing.high, *tmp)
```

Syntax example:

```
do_pack(options:pack_options, l:*list of bit) is only {
    var L : list of bit = pack(packing.low, operand2,
        operand1, operand3);

    l.add(L)
}
```

#### 20.4.1.1.2 do\_unpack()

<b>Purpose</b>	Unpack a packed list of bit into a struct
<b>Category</b>	Predefined method of <b>any_struct</b>
<b>Syntax</b>	<b>do_unpack</b> ( <i>options</i> :pack options, <i>l</i> : list of bit, <i>from</i> : int): int
<b>Parameters</b>	<i>options</i> This parameter is an instance of the <b>pack_options</b> struct (see <a href="#">20.3</a> ).
	<i>l</i> A list of bits containing data to be stored in the struct fields.
	<i>from</i> An integer that specifies the index of the bit to start unpacking.
	int (return value) An integer that specifies the index of the last bit in the list of bits that was unpacked.

The **do\_unpack()** method is called automatically whenever data is unpacked into the current struct. This method unpacks bits from a list of bits into the physical fields of the struct. It starts at the bit with the specified index, unpacks in the order defined by the pack options, and fills the current struct's physical fields in the order they are defined.

This method returns an integer, which is the index of the last bit unpacked into the list of bits.

The method shall issue a runtime error message if the struct has no physical fields. If there are leftover bits at the end of packing, it is not an error. If more bits are needed than currently exist in the list of bits, a runtime error shall be issued.

The **do\_unpack()** method can be extended to create a unique unpacking scenario for that struct by creating a custom **pack\_options** instance (see [20.3](#)).

The following considerations also apply:

- Do not call the **do\_unpack()** method of any struct directly, e.g., `my_struct.do_unpack( )`. Use **unpack()** instead, e.g., `unpack(packing.high, lob, my_struct)`.
- When the **do\_unpack()** method is modified, the index of the last bit in the list of bits that was unpacked also needs to be calculated and returned.

#### Example

The following call to `unpack( )`

```
unpack(packing.low, lob, c1, c2)
```

makes the following calls to the **do\_unpack** method of each struct:

```

c1.do_unpack(packing.low, lob, index);
c2.do_unpack(packing.low, lob, index)

```

Syntax example:

```

do_unpack(options:pack_options, l:list of bit, src:int): int is only {
    var L : list of bit = l[src..];

    unpack(packing.low, L, op2, op1, op3);
    return src + 8 + 5 + 3           //bit-width of operands
}

```

#### 20.4.1.2 A simple example of packing

This example shows how packing converts data from a struct into a stream of bits. An instruction struct is defined as:

```

struct instruction {
    %opcode      : uint (bits:3);
    %operand     : uint (bits:5);
    %address     : uint (bits:8);
    !data_packed_low : list of bit;

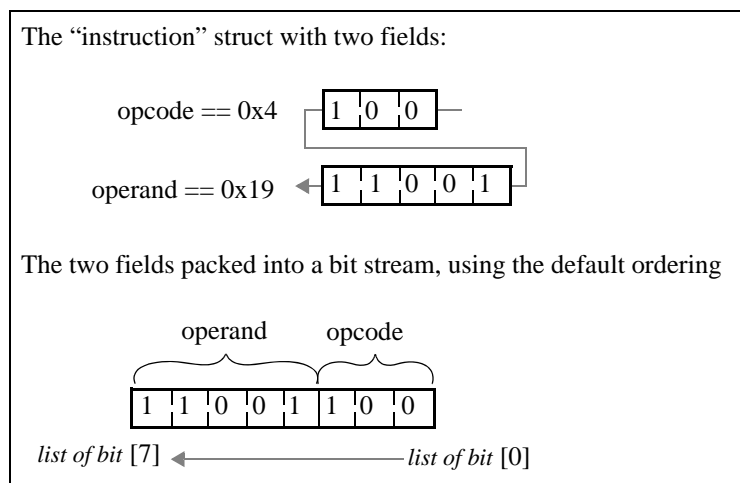
    keep opcode == 0b100;
    keep operand == 0b11001;
    keep address == 0b00001111
}

```

The **post\_generate()** method of this struct is extended to pack the opcode and the operand fields into two variables. The order in which the fields are packed is controlled with the packing option parameter.

```
data_packed_low = pack(NULL, opcode, operand)
```

When NULL is passed as the packing option parameter, the LSB of the first expression in **pack()**, opcode, is placed at index [0] in the resulting list of bit. The MSB of the last expression, operand, is placed at the highest index in the resulting list of bit. [Figure 13](#) shows this packing order.



**Figure 13—Simple packing example showing packing.low**



### 20.4.1.3 A simple example of unpacking

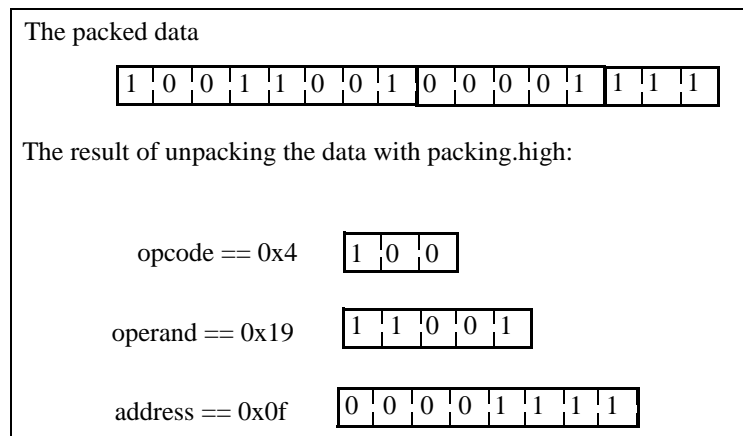
This example shows how packing fills the fields of a struct instance with data from a bit stream. An instruction struct is defined as:

```
struct instruction {
    %opcode : uint (bits:3);
    %operand : uint (bits:5);
    %address : uint (bits:8)
}
```

The extension to **post\_generate()** shown as follows unpacks a list of bits, `packed_data`, into a variable `inst` of type `instruction` using the **packing.high** option. The results are shown in [Figure 14](#).

```
extend sys {
    post_generate() is also {
        var inst : instruction;
        var packed_data : list of bit;

        packed_data = {1;1;1;1;0;0;0;0;1;0;0;1;1;0;0;1};
        unpack(packing.high, packed_data, inst)
    }
}
```



**Figure 14—Simple unpacking example showing `packing.high`**

In this case, the expression that provides the value, `packed_data`, is a list of bits. When a value expression is not a list of bits, *e* uses implicit packing to store the data in the target expression (see [20.5](#)).

### 20.4.2 Packing and unpacking lists

Packing a list creates a bit stream by concatenating the list items together, starting with the item at index [0].

Unpacking a bit stream into a list fills the list item-by-item, starting with the item at index [0]. The size of the unpacked list is determined by whether the list is sized and whether it is empty.

- Unpacking into an empty list expands the list as needed to contain all the available bits.
- Unpacking into a non-empty list unpacks only until the existing list size is reached.
- Unpacking to a struct fills the sized lists only to their defined size, regardless of their actual size at the time.

- Unpacking into an unsized, uninitialized list shall cause a runtime error message, because the list is expanded as needed to consume all the given bits.

NOTE—When a struct is allocated, the lists within it are empty. If the lists are sized, unpacking is performed until the defined size is reached.

See [Clause 4](#) for more information on sizing lists.

#### *Example*

This example shows the recommended way to get a variable number of list items. The specification order is important because the `len1` and `len2` values need to be set before initializing `data1` and `data2`. Declaring `len1` and `len2` before `data1` and `data2` ensures the list length is generated first. Unpacking into a list with a variable number of items requires packing and passing the number of items in the list before unpacking the list.

```
struct packet {
    %len1          : int;
    %len2          : int;
    %data1[%len1]   : list of byte;
    %data2[%len1 + %len2] : list of byte
}
```

### 20.4.3 Packing and unpacking scalar expressions

Packing a scalar expression creates an ordered bit stream by concatenating the bits of the expression together. Unpacking a bit stream into a scalar expression fills the scalar expression by putting the lowest bit of the bit stream into the lowest bit of the scalar expression. If a list is unpacked into one or more scalar expressions and there are not enough bits in the list to put a value into each scalar, a runtime error shall be issued.

Packing and unpacking of a scalar expression is performed using the expression's inherent size, except when the expression contains a bit-slice operator. Missing bits are assumed to be zero (0) and extra bits are allowed (and ignored). See also [5.1](#).

The bit-slice operator `[ : ]` can also be used to select a subrange of an expression to be packed or unpacked. This operator does not change the type of the pack or unpack expression.

### 20.4.4 Packing and unpacking strings

Packing a string creates an ordered bit stream by concatenating each ASCII byte of the string together from left-to-right, ending with a byte with the value zero (the final NULL byte). Unpacking a string places the bytes of the string into the target expression, starting with the first ASCII byte in the string up to and including the first byte with the value zero (0).

The `as_a()` method, which converts directly between the **string** and **list of byte** types, can also be used to obtain different results (see [5.8.1](#)).

### 20.4.5 Packing values for real types

Real values take up 64 bits when packed. These bits are actual bit representation of the double value. The effect of the various packing options on real type objects is similar to their effect on an integer (bits:64) value.

## 20.5 Implicit packing and unpacking

Implicit packing and unpacking is always performed using the parameters of **packing.low** as follows.

- When an untyped expression is assigned to a scalar or list of scalars, it is implicitly unpacked before it is assigned. Untyped expressions include HDL signals, pack expressions, and bit concatenations (see [5.2](#)).

```
var my_list : list of int = {1; 2; 3};  
var int_10 : int(bits:10);
```

```
my_list = 'top.foo';  
int_10 = pack(NULL, 5)
```

- When a scalar or list of scalars is assigned to an untyped expression, it is implicitly packed before it is assigned:

```
'top.foo' = {1; 2; 3}
```

- When the value expression of an unpack action is other than a list of bits, it is implicitly packed before it is unpacked:

```
unpack(packing.low, 5, my_list)
```

Implicit packing and unpacking is not supported for strings, structs, or lists of non-scalar types.



## 21. Control flow actions

This clause describes the *e* control flow actions.

### 21.1 Conditional actions

*Conditional actions* are used to specify code segments that execute only when a specific condition is met.

#### 21.1.1 if then else

<b>Purpose</b>	Perform an action block if a given Boolean expression is TRUE or a different action if the expression is FALSE
<b>Category</b>	Action
<b>Syntax</b>	<b>if</b> <i>bool-exp</i> [ <b>then</b> ] { <i>action</i> ; ...} [ <b>else if</b> <i>bool-exp</i> [ <b>then</b> ] { <i>action</i> ; ...}] [ <b>else</b> { <i>action</i> ; ...}]
<b>Parameters</b>	<i>bool-exp</i> A Boolean expression.
	<i>action</i> ; ...                      A list of zero or more actions separated by semicolons (;) and enclosed in braces ({}).

If the first *bool-exp* is TRUE, the **then** *action* block is executed. If the first *bool-exp* is FALSE, the **else if** clauses are executed sequentially: if an **else if** *bool-exp* is found that is TRUE, its **then** *action* block is executed; otherwise, the final **else** *action* block is executed.

Because **if then else** is a single action, no semicolons (;) should appear between **if** and **else**, unless they are required to separate two or more actions within one of the action blocks.

NOTE—Since **else if then** clauses can be used for multiple Boolean checks (comparisons), consider using a **case bool-case-item** action (see [21.1.3](#)) when there are a large number of comparisons to perform.

Syntax example:

```

if a > b then {
    print a, b
} else if a == b then {
    print a
} else {
    print b, a
}

```

### 21.1.2 case labeled-case-item

<b>Purpose</b>	Execute an action block based on whether a given comparison is TRUE	
<b>Category</b>	Action	
<b>Syntax</b>	<b>case</b> <i>case-exp</i> { <i>labeled-case-item</i> ; ... [ <b>default:</b> { <i>default-action</i> ; ... }] }	
<b>Parameters</b>	<i>case-exp</i>	A legal <i>e</i> expression.
	<i>labeled-case-item</i>	<i>label-exp</i> [:] <i>action-block</i> where <i>label-exp</i> is a value or a range <i>action-block</i> is a list of zero or more actions separated by semicolons and enclosed in braces. Syntax: { <i>action</i> ; ... } The entire <i>labeled-case-item</i> is repeatable, not just the <i>action-block</i> related to the <i>label-exp</i> .
	<i>default-action</i> ; ...	A list of zero or more actions separated by semicolons ( ; ) and enclosed in braces ( { } ).

This evaluates the *case-exp* and executes the first *action-block* for which *label-exp* matches the *case-exp*. If no *label-exp* equals the *case-exp*, it executes the *default-action* block, if specified.

After an *action-block* is executed, the *e* program proceeds to the line that immediately follows the entire **case** statement.

Syntax example:

```

case packet.length {
    64          : {out("minimal packet")      };
    [65..256]   : {out("short packet")        };
    [257..512]  : {out("long packet")         };
    default     : {out("illegal packet length")}
}

```

### 21.1.3 case bool-case-item

<b>Purpose</b>	Execute an action block based on whether a given comparison is TRUE
<b>Category</b>	Action
<b>Syntax</b>	<b>case</b> { <i>bool-case-item</i> ; ... [ <b>default</b> { <i>default-action</i> ; ... }] }
<b>Parameters</b>	<i>bool-case-item</i> <i>bool-exp</i> [:] <i>action-block</i> where <i>bool-exp</i> is a Boolean expression <i>action-block</i> is a list of zero or more actions separated by semicolons and enclosed in braces. Syntax: { <i>action</i> ; ... } The entire <i>bool-case-item</i> is repeatable, not just the <i>action-block</i> related to the <i>bool-exp</i> .
	<i>default-action</i> ; ...    A list of zero or more actions separated by semicolons ( ; ) and enclosed in braces ( { } ).

This evaluates the *bool-exp* conditions one after the other and executes the *action-block* associated with the first TRUE *bool-exp*. If no *bool-exp* is TRUE, it executes the *default-action-block*, if specified. After an *action-block* is executed, the *e* program proceeds to the line that immediately follows the entire case statement.

Each of the *bool-exp* conditions is independent of the other *bool-exp* conditions and there is no main *case-exp* to which all cases refer, unlike the case labeled-case-item (see [21.1.2](#)).

NOTE—This case action has the same functionality as a single **if then else** action, where each *bool-case-item* is specified as a separate **else if then** clause.

Syntax example:

```
case {
  packet.length == 64           {out("minimal packet")};
  packet.length in [65..255] {out("short packet") };
  default                       {out("illegal packet") }
}
```

## 21.2 Iterative actions

*Iterative actions* are used to specify code segments that execute in a loop, for multiple times, in a sequential order.

NOTE 1—A **repeat until** action performs the action block at least once. A **while** action might not perform the action block at all.

NOTE 2—The optional **do** syntax used in some of the constructs of this subclause is purely syntactic sugar.

### 21.2.1 while

<b>Purpose</b>	Execute an action block repeatedly as long as a Boolean expression evaluates to TRUE	
<b>Category</b>	Action	
<b>Syntax</b>	<b>while</b> <i>bool-exp</i> [ <b>do</b> ] { <i>action</i> ; ...}	
<b>Parameters</b>	<i>bool-exp</i>	A Boolean expression.
	<i>action</i> ; ...	A list of zero or more actions separated by semicolons (:) and enclosed in braces ({}).

This executes the *action* block repeatedly in a loop while *bool-exp* is TRUE. This construct can be used to set up a perpetual loop as `while TRUE {}`. The loop shall not execute at all if the *bool-exp* is FALSE when the **while** action is encountered.

Syntax example:

```
while a < b do {
    a += 1
}
```

### 21.2.2 repeat until

<b>Purpose</b>	Execute an action block repeatedly as long as a Boolean expression evaluates to FALSE	
<b>Category</b>	Action	
<b>Syntax</b>	<b>repeat</b> { <i>action</i> ; ...} <b>until</b> <i>bool-exp</i>	
<b>Parameters</b>	<i>action</i> ; ...	A list of zero or more actions separated by semicolons (:) and enclosed in braces ({}).
	<i>bool-exp</i>	A Boolean expression.

This executes the *action* block repeatedly in a loop until *bool-exp* is TRUE. The action block is executed at least once.

Syntax example:

```
repeat {
    i += 1
} until i == 3
```



### 21.2.3 for each in

<b>Purpose</b>	Execute an action block once for every element of a list expression
<b>Category</b>	Action
<b>Syntax</b>	<b>for each</b> [ <i>type</i> ] [( <i>item-name</i> )] [ <b>using index</b> ( <i>index-name</i> )] <b>in</b> [ <b>reverse</b> ] <i>list-exp</i> [ <b>do</b> ] { <i>action</i> ; ...}
<b>Parameters</b>	<i>type</i> A type of the struct comprising the list specified by <i>list-exp</i> . Elements in the list shall match this type.
	<i>item-name</i> The name of the current item in <i>list-exp</i> . If this parameter is not specified, the item can be referenced using the implicit variable <b>it</b> .
	<i>index-name</i> The name of the index of the current list item. If this parameter is not specified, the item can be referenced using the implicit variable <b>index</b> .
	<i>list-exp</i> An expression that results in a list.
	<i>action</i> ;...                A list of zero or more actions separated by semicolons (;) and enclosed in braces ({}).

For each item in *list-exp*, if its type matches *type*, this executes the *action* block. Inside the *action* block, the implicit variable **it** (if no *item-name* is specified) or the optional *item-name* (when specified) refers to the matched item, and the implicit variable **index** (or the optional *index-name*) reflects the index of the current item. If **reverse** is specified, *list-exp* is traversed in reverse order, from last to first. The implicit variable **index** (or the optional *index-name*) starts at zero (0) for regular loops and at `list.size() - 1` for reverse loops.

Each **for each in** action defines two new local variables for the loop, named by default **it** and **index**. The following restrictions also apply:

- d) When loops are nested inside one another, the local variables of the internal loop hide those of the external loop. To overcome this, assign each *item-name* and *index-name* unique names.
- e) Within the action block, a value cannot be assigned to **it** or **index**—or to *item-name* or *index-name*.

Syntax example:

```
for each transmit packet (tp) in sys.pkts do {
    print tp           // "transmit packet" is a type
}
```

### 21.2.4 for from to

<b>Purpose</b>	Execute a for loop for the number of times specified
<b>Category</b>	Action
<b>Syntax</b>	<b>for</b> <i>var-name</i> <b>from</b> <i>from-exp</i> [ <b>down</b> ] <b>to</b> <i>to-exp</i> [ <b>step</b> <i>step-exp</i> ] [ <b>do</b> ] { <i>action</i> ; ...}
<b>Parameters</b>	<i>var-name</i> A temporary variable of type <b>int</b> .
	<i>from-exp</i> , <i>to-exp</i> , <i>step-exp</i> Valid <i>e</i> expressions that resolve to type <b>int</b> . The default value for <i>step-exp</i> is 1.
	<i>action</i> ; ...            A list of zero or more actions separated by semicolons ( ; ) and enclosed in braces ( { } ).

This creates a temporary variable *var-name* of type **int** and repeatedly executes the *action* block while incrementing (or decrementing if **down** is specified) its value from *from-exp* to *to-exp* in interval values specified by *step-exp* (which defaults to 1), i.e., the loop is executed until the value of *var-name* is greater than the value of *to-exp* or less than *to-exp*.

The temporary variable *var-name* is visible only within the **for from to** loop where it was created.

Syntax example:

```
for i from 5 down to 1 do {
    out(i)
}           // Outputs 5,4,3,2,1
```

### 21.2.5 for

<b>Purpose</b>	Execute a C-style for loop
<b>Category</b>	Action
<b>Syntax</b>	<b>for</b> { <i>initial-action</i> ; <i>bool-exp</i> ; <i>step-action</i> } [ <b>do</b> ] { <i>action</i> ; ...}
<b>Parameters</b>	<i>initial-action</i> An action.
	<i>bool-exp</i> A Boolean expression
	<i>step-action</i> An action.
	<i>action</i> ;...            A list of zero or more actions separated by semicolons ( ; ) and enclosed in braces ( { } ).

The **for** loop works similarly to the `for` loop in the C language. This **for** loop executes the *initial-action* once and then checks the *bool-exp*. If the *bool-exp* is TRUE, it executes the *action* block followed by the *step-action*. It repeats this sequence in a loop for as long as *bool-exp* is TRUE. The following restrictions also apply:

- When a loop variable is used within a **for** loop, it needs to be declared before the loop (unlike the temporary variable of type **int** automatically declared in a **for from to** loop).

- b) Although this action is similar to a C-style `for` loop, the *initial-action* and *step-action* need to be *e* style actions.

Syntax example:

```
for i from 5 down to 1 do {
    out(i)
} // Outputs 5,4,3,2,1
```

## 21.3 File iteration actions

This subclause describes *loop constructs*, which are used to manipulate general ASCII files.

### 21.3.1 for each line in file

<b>Purpose</b>	Iterate a for loop over all lines in a text file
<b>Category</b>	Action
<b>Syntax</b>	<b>for each [line] [(name)] in file</b> <i>file-name-exp</i> [ <b>do</b> ] { <i>action</i> ; ...}
<b>Parameters</b>	<i>name</i> Variable referring to the current line in the file.
	<i>file-name-exp</i> A string expression that gives the name of a text file.
	<i>action</i> ; ... A list of zero or more actions separated by semicolons (;) and enclosed in braces ({ }).

This executes the *action* block for each line in the text file *file-name*. Inside the block, **it** (if no name is specified) or *name* (if specified) refers to the current line (as a string) without the final line feed (`\n`) character.

NOTE—The optional **line** syntax is purely syntactic sugar.

Syntax example:

```
for each line in file "signals.dat" do {
    '(it)' = 1
} // Reads a list of signal names and assigns to each the value 1
```

### 21.3.2 for each file matching

<b>Purpose</b>	Iterate a for loop over a group of files	
<b>Category</b>	Action	
<b>Syntax</b>	<b>for each file</b> [( <i>name</i> )] <b>matching</b> <i>file-name-exp</i> [ <b>do</b> ] { <i>action</i> ; ...}	
<b>Parameters</b>	<i>name</i>	Variable referring to the current line in the file.
	<i>file-name-exp</i>	A string expression that gives the name of a text file.
	<i>action</i> ; ...	A list of zero or more actions separated by semicolons (;) and enclosed in braces ({}).

For each file (in the file search path) whose name matches *file-name-exp*, this executes the *action* block. Inside the block, **it** (if no *name* is specified) or *name* (if specified) refers to the matching file name.

Syntax example:

```
for each file matching "*.e" do {
    out(it)
} // lists the 'e' files in the current directory
```

## 21.4 Actions for controlling the program flow

These actions alter the flow of the program in places where the flow would otherwise continue differently.

### 21.4.1 break

<b>Purpose</b>	Break the execution of a loop
<b>Category</b>	Action
<b>Syntax</b>	<b>break</b>

This breaks the execution of the nearest enclosing iterative action (**for** or **while**). When a **break** action is encountered within a loop, the execution of actions within the loop is terminated and the next action to be executed is the first one following the loop.

**break** actions shall not be placed outside the scope of a loop (or the compiler shall report an error).

Syntax example:

```
break
```

### 21.4.2 continue

<b>Purpose</b>	Stop executing the current loop iteration and start executing the next loop iteration
<b>Category</b>	Action
<b>Syntax</b>	<b>continue</b>

This stops the execution of the nearest enclosing iteration of a **for** or **while** loop, and continues with the next iteration of the same loop. When a **continue** action is encountered within a loop, the current iteration of the loop is aborted, and execution continues with the next iteration of the same loop.

**continue** actions shall not be placed outside the scope of a loop (or the compiler shall report an error).

Syntax example:

```
continue
```



## 22. Importing and preprocessor directives

This clause contains the following sections:

- a) **Importing *e* modules**—Defines how to use the **import** statement.
- b) **Importing *e* modules**—Use these preprocessor directives to control *e* processing. The preprocessor directives check for the existence of a **#define** for a given name.
  - 1) **#ifdef**—If a given name is defined, use the attached code; otherwise, use different code.
  - 2) **#ifndef**—If a given name is not defined, use the attached code; otherwise, use different code.
 The **#ifdef** and **#ifndef** directives can be used as statements, struct members, or actions.
- c) **#define**—Defines a preprocessor replacement rule (or flag).
- d) **#undef**—Removes the definition of the replacement rule (or flag).

See also [Clause 16](#) and [Annex B](#).

### 22.1 Importing *e* modules

Import statements declare dependency between two modules (*e* files). An **import** statement ensures entities (types, methods, fields, etc.) declared in the imported module are available in the importing module. See also [4.1.1](#).

#### 22.1.1 import

<b>Purpose</b>	Declare dependency of the current module on others.
<b>Category</b>	Statement
<b>Syntax</b>	<b>import</b> <i>file-name</i> , ...   ( <i>file-name</i> , ... )
<b>Parameters</b>	<div> <i>file-name</i>, ...           </div> <div>             One or more names of <i>e</i> files, separated by commas ( , ). If no extension is given for a file name, an .e extension is assumed. File names can contain references to environment variables using the POSIX-style notation <i>\$name</i> or <i>\${name}</i>. The relative path indicators <i>.</i> / and <i>.</i> . / can be used in file names. Enclosing the file-name list in parentheses ( ( ) ) signifies explicit cyclic import (see <a href="#">22.1.2</a>).           </div>

Source files in *e* are taken as modules. A module bears the name of the file from which it was read (without the extension) and this name serves to identify it.

An *e* environment can load a module only once. Thus, an **import** statement shall cause the imported module to be loaded into the environment only if no module by that name is already loaded.

When a module is required by an **import** statement and that module is not already loaded, the file is searched for in the file system according to some (implementation-dependent) set of search priorities.

**import** statements determine not only which modules need to be loaded, but also the order in which they are loaded. The exact order can have implications on the semantics of the program. See also [B.4](#).

Syntax example:

```
import test_drv.e
```

### 22.1.2 Cyclic referencing and importing

*e* allows mutually dependent definitions both within a single module and between different modules (forward referencing). A *dependency unit* is a portion of code in which use can be made of entities declared anywhere within it. By default, each module (source file) is a dependency unit by itself. When the definitions in one module presuppose entities declared in another and vice versa, both modules shall belong to a single dependency unit. The same holds for cyclic dependencies of any number of modules.

**import** statements are used to handle cyclic dependencies between modules either implicitly or explicitly. In both cases, code anywhere inside the set of modules can make use of entities declared anywhere else in these modules.

- *Implicit cycles* in the import graph are cases where module  $m_1$  imports module  $m_2$ , which imports  $m_3$ , ... and module  $m_n$  imports  $m_1$  again. All of these are taken as a single dependency unit.
- *Explicit cycles* are declared by an **import** statement with a list of modules enclosed in parentheses ( ( ) ). In this case, the modules stated are taken as a dependency unit on which the current module depends.

See [B.3](#) for more on cyclic importing.

### 22.2 #ifdef, #ifndef

<b>Purpose</b>	Conditionally include a set of constructs at compile-time	
<b>Category</b>	Statement, struct member, action	
<b>Syntax</b>	<b>#if[n]def</b> [ <i>name</i> ] <b>[then]</b> { <i>e-code</i> } <b>[#else { <i>e-code</i> }]</b>	
<b>Parameters</b>	<i>name</i>	An <i>e</i> identifier. When preceded by backtick ( ` ), a Verilog-style define is intended.
	<i>e-code</i>	<i>e</i> code to be included, based on whether the preprocessor define has been defined. <ul style="list-style-type: none"> <li>— For an <b>#ifdef</b> or <b>#ifndef</b> statement, only <i>e</i> statements can be used in <i>e-code</i>.</li> <li>— For an <b>#ifdef</b> or <b>#ifndef</b> struct member, only struct members can be used in <i>e-code</i>.</li> <li>— For an <b>#ifdef</b> or <b>#ifndef</b> action, only actions can be used in <i>e-code</i>.</li> </ul>

The **#ifdef** and **#ifndef** preprocessor directives can be used along with **#define** to cause the *e* parser to process particular code or ignore it, depending on whether a given name has been defined.

- The **#ifdef** syntax checks whether the name has been defined and, if it has, includes the code inside the (first) braces ( { } ).
- The **#ifndef** syntax checks whether the name has been defined and if it has not, includes the code inside the (first) braces ( { } ).

The optional **#else** syntax provides an alternative statement when the condition of the **#ifdef** or **#ifndef** does not hold.

- For **#ifdef**, if the name has not been defined, the **#else** code is included.
- For **#ifndef**, if the name has been defined, the **#else** code is included.



Syntax example:

```
#ifdef MEM_LG then {
    import mml.e
}
```

## 22.3 #define

<b>Purpose</b>	Define a preprocessor flag and replacement rule	
<b>Category</b>	Statement	
<b>Syntax</b>	<b>[#]define</b> [ <b>`</b> ] <i>name</i> [ <i>replacement</i> ]	
<b>Parameters</b>	<i>name</i>	Any legal <i>e</i> identifier (case-sensitive). A backtick ( <b>`</b> ) prefix signifies a Verilog-style <code>define</code> . The backtick is taken as part of the defined name (i.e., defining <code>`X</code> is not equivalent to defining <code>X</code> ).
	<i>replacement</i>	Any syntactic element, for example, an expression or an HDL variable. This replaces the name wherever the name appears in the <i>e</i> code that follows the <b>#define</b> statement.

The defined name functions both as a preprocessor flag and replacement rule. As a flag, it affects subsequent conditional compilation directives (**#ifdef** and **#ifndef**). As a replacement rule, the given text is substituted for every occurrence of the name in subsequent code, except inside string literals.

A **#define** statement applies only to code that comes later with respect to the preprocessing order. In particular, a **#define** statement before an **import** statement might affect the code in the imported module. But it does not affect it when the module is already loaded or in some cases of cyclic import. See also [B.5](#).

The *replacement* in a **#define** statement can contain line breaks (newline characters) only if preceded by backslash (**\**). The *replacement* shall not contain the *name*; if it does, this shall result in a runtime error.

Use parentheses around the *replacement* when they are needed to ensure proper associativity, e.g.,

```
define LX len + m
```

is different than:

```
define LX (len + m)
```

In an expression like `lenx = 2*LX`, the first case becomes `lenx = 2*len + m`, while the second case becomes `lenx = 2*(len+m)`.

Syntax example:

```
#define PLIST_SIZE 100
```

## 22.4 #undef

<b>Purpose</b>	Undefine a preprocessor rule	
<b>Category</b>	Statement	
<b>Syntax</b>	<b>#undef</b> [ <b>`</b> ] <i>name</i>	
<b>Parameters</b>	<i>name</i>	An <i>e</i> identifier. When preceded by backtick ( <b>`</b> ), a Verilog-style <code>define</code> is intended.

This removes a preprocessor rule that was defined previously by a **#define** statement. The name is not recognized by the preprocessor from that point onward. Just like **#define** statements, **#undef** statements apply to code that comes later with respect to the preprocessing order. An **#undef** statement before an **import** statement might affect the code in the imported module and other modules imported by it. See also [B.5](#).

The following rules also apply:

- A preprocessor rule that is undefined in a compiled *e* module is not accessible to the C interface.
- A preprocessor rule that has been undefined can be redefined later, with any value. The last value is accessible to the C interface.
- If the undefined preprocessor rule was not previously defined, **#undef** has no effect.

Syntax example:

```
#undef PLIST_SIZE
```

## 23. Encapsulation constructs

This clause contains the syntax and descriptions of the *e* statements used to create packages and modify access control.

### 23.1 package: package-name

<b>Purpose</b>	Associates a module with a package
<b>Category</b>	Statement
<b>Syntax</b>	<b>package</b> <i>package-name</i>
<b>Parameters</b>	<i>package-name</i> A standard <i>e</i> identifier that assigns a unique name to the package. It is legal for a package name to be the same as a module or type name.

This associates a module with a package. Only one **package** statement can appear in a file and it shall be the first statement in the file. A file with no **package** statement is equivalent to a file beginning with the **package main** statement (**main** is the default package).

Syntax example:

```
package vr_xb
```

### 23.2 package: type-declaration

<b>Purpose</b>	Modifies access to a type or a struct
<b>Category</b>	Statement
<b>Syntax</b>	<b>[package]</b> <i>type-declaration</i>
<b>Parameters</b>	<i>type-declaration</i> An <i>e</i> type declaration (for a struct, unit, enumerated list, or other type).

The **package** modifier can be used to shield the defined struct member from code outside the package files. This includes declaring a variable of the type, extending, inheriting, casting using the **as\_a** operator, and all other contexts in which the name of a type is used. It is equivalent to the default (package) access level for classes in Java.

- The definition of a **when** subtype (using a **when** or **extend** clause) does not allow for an access modifier. A **when** subtype is public, unless its base struct or one of its determinant fields is declared **package**. A **when** subtype cannot have a **private** or **protected** determinant field.
- Any reference to such a **when** subtype (i.e., with a **private** or **protected** determinant field), even in a context in which the **when** determinant field is accessible, shall result in a compilation error.
- A **package** type defined in the **main** package is visible only inside the **main** package; this has no bearing on what the global visibility is when no **package** restriction is specified.

Without the **package** modifier, the type or struct has no access restriction.

See also [D.3.2](#) and [D.3.3](#).

Syntax example:

```
package type t : int(bits:16)
```

### 23.3 package | protected | private: struct-member

<b>Purpose</b>	Modifies access to a struct field, method, or event	
<b>Category</b>	Keyword	
<b>Syntax</b>	<b>package</b> <i>struct-member-definition</i> <b>protected</b> <i>struct-member-definition</i> <b>private</b> <i>struct-member-definition</i>	
<b>Parameters</b>	<i>struct-member-definition</i>	A struct or unit field, method, or event definition. See <a href="#">Clause 6</a> for the syntax of struct and unit member definitions.

A struct member declaration can include a **package**, **protected**, or **private** keyword to modify access to the struct member. If no access modifier exists in the declaration of a struct member, the struct member has no access restriction (the default is public).

- The **package** modifier means code outside the package files cannot access the struct member. It is equivalent to the default (package) access level for fields and methods in Java.
- The **protected** modifier means code outside the struct family scope cannot access the struct member. It is similar (although not equivalent) to the *protected* semantics in other OO languages.

The *struct family scope* is the code within the definition of a struct (declaration and extensions), as well as the definition of all **when** and **like** subtypes.

- The **private** modifier means only code within both the package and the struct family scope can access the struct member. This means code within the extension of the same struct in a different package is outside its accessibility scope.

This modifier is less restrictive than the *private* attribute of other OO languages, as methods of derived structs or units within the same package can access a private struct member.

Only fields, methods, and events can have access restrictions. There are other named struct members in *e*, namely **cover groups** and named **expects**, to which access control does not apply—they are completely public. However, **cover groups** and **expects** are defined in terms of fields, methods, and events, and can refer to other entities in their definitions according to the accessibility rules.

- a) An extension of a struct member can restate the same access modifier as the declaration has or omit the modifier altogether. If a different modifier appears, the compiler shall issue an error.
- b) All references to a struct member outside its accessibility scope shall result in an error at compile time. Using an enumerated field's value as a **when** determinant is considered such a reference, even if the field name is not explicitly mentioned.
- c) If the type of a field or the return type or any parameter type of a method has **package** access and the field or method is a member of a struct that is not declared **package**, the field or method shall be explicitly declared **package** or **private**. If the field or method is not declared **package** or **private** under these conditions, the compiler shall issue an error.

Syntax examples:

```
private f : int;
protected m() is {};
package event e
```

## 23.4 Scope operator (::)

<b>Purpose</b>	Identify the package scope for the given type reference
<b>Category</b>	Special purpose operator
<b>Syntax</b>	<i>package-name</i> :: <i>type-name</i>
<b>Parameters</b>	<i>package-name</i> The name of the package where the type was declared.
	<i>type-name</i> The name of a struct, unit, or scalar type.

The *scope operator* qualifies the name for a given struct, unit, or scalar type by defining the package where the type belongs. This is required if it would be unclear which type is being referenced. Each type referenced is evaluated according to the type-scoping rules; an error shall occur if the reference is ambiguous. See also [Annex D](#).

Syntax example:

```
xbus_evc : vr_xbus::env_u
```



## 24. Simulation-related constructs

This clause describes simulation-related actions, expressions, and routines. See also [Clause 9](#).

Simulators can be attached to the *e* runtime environment by means of a simulator adapter. In addition, it may be necessary to infer support structures, such as extra registers, to facilitate the integration. Such structures are placed in a *stubs file*, which can be generated by the *e* compiler. The simulation interface and stub file generation functions are implementation-dependent.

### 24.1 force

<b>Purpose</b>	Force a value on an HDL object
<b>Category</b>	Action
<b>Syntax</b>	<b>force</b> ' <i>HDL-pathname</i> ' = <i>exp</i>
<b>Parameters</b>	<i>HDL-pathname</i> The full pathname of an HDL object (see <a href="#">24.3</a> ), including any expressions.
	<i>exp</i> Any scalar expression or literal constant, as long as it is composed only of 1's and 0's. No x or z values are allowed. Thus, 16'hf0f1 or <code>sys.my_val+5</code> are legal values.

This forces an HDL object to a specified value, overriding the current value and preventing the DUT from driving any value. The HDL object remains at the specified value until a subsequent **force** action from *e* or until freed by a **release** action (see [24.2](#)). The following also apply:

- If part of a vectored object is forced, the **force** action is propagated to the rest of the object.
- To force single elements of an array of a scalar integer or enumerated type, use the predefined routine **simulator\_command()** (see [24.4](#)).

Syntax example:

```
force '~/top/sig' = 7
```

### 24.2 release

<b>Purpose</b>	Remove a <b>force</b> action from an HDL object
<b>Category</b>	Action
<b>Syntax</b>	<b>release</b> ' <i>HDL-pathname</i> '
<b>Parameters</b>	<i>HDL-pathname</i> The full pathname of an HDL object previously specified in a <b>force</b> action.

This releases the HDL object that previously has been forced (see [24.1](#)).

Syntax example:

```
release 'top.sig'
```

### 24.3 Tick access: 'hdl-pathname'

<b>Purpose</b>	Access HDL objects, using <i>full-path-names</i>
<b>Category</b>	Expression
<b>Syntax</b>	<b>'HDL-pathname[index-exp   bit-range] [@(x   z   n)]'</b>
<b>Parameters</b>	<i>HDL-pathname</i> The full pathname of an HDL object, including any expressions and composite data.
	<i>index-exp</i> Accesses a single bit of a Verilog vector, a single element of a Verilog memory, or a single vector of a VHDL array of vectors.
	<i>bit-range</i> <i>bit-range</i> has the format [ <i>high-bit-num</i> : <i>low-bit-num</i> ] and is extracted from the object from the high bit to low bit. Slices of buses are treated exactly as they are in HDL languages. They need to be specified in the same direction as in the HDL code and reference the same bit numbers.
	<b>@x   @z</b> Sets or gets the x or z component of the value. When this notation is not used in accessing an HDL object, <i>e</i> translates the values of x to zero (0) and z to one (1). When reading HDL objects using @x (or @z), <i>e</i> translates the specified value (x or z) to one (1) and all other values to zero (0). When writing HDL objects, if @x (or @z) is specified, <i>e</i> sets every bit that has a value of 1 to x (or z). In this way, @x or @z acts much like a data mask, manipulating only those bits that match the value of x or z.
	<b>@n</b> When this specifier is used for driving HDL objects, the new (simulator) value is visible immediately (now). The <i>default mode</i> is to buffer projected values and update only at the end of the tick.

This accesses Verilog and VHDL objects from *e*.

Syntax example:

```
'~/top/sig' = 7;
print '~/top/sig'
```

### 24.4 simulator\_command()

<b>Purpose</b>	Issue a simulator command
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>simulator_command(command: string)</b>
<b>Parameters</b>	<i>command</i> A valid simulator command, enclosed in double quotes (" "). <b>simulator_command()</b> cannot be used to pass commands that change the state of simulation, such as run, restart, restore, or exit (to the simulator).

This passes a command to the HDL simulator from *e*. The command shall not return a value. The output of the command is sent to the standard output and log file.



Syntax example:

```
simulator_command("force -deposit memA(31:0) ")
```

## 24.5 stop\_run()

<b>Purpose</b>	Stop a simulation run cleanly
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>stop_run()</b>

This stops the simulator and initiates post-simulation phases. This method needs to be called by a user-defined method or TCM to stop the simulation run cleanly. The following things occur when **stop\_run()** is invoked:

- The **quit()** method of each struct under **sys** is called. Each **quit()** method emits a “quit” event for that struct instance at the end of the current tick.
- All executing threads shall continue until the end of the current tick.
- At the end of the current tick, the extract, check, and finalize test phases are performed.
- If a simulator is linked here, *e* terminates the simulation cleanly after the test is finalized.

Plus, the following restrictions also apply:

- Executing a tick after calling **stop\_run()** shall be considered an error.
- If the simulator `exit` command is called before **stop\_run()**, the global methods for extracting, checking, and finalizing the test are called.

NOTE—Use **sys.extract()** and extend that to make something happen right after stopping a run (rather than extending or modifying the **stop\_run()** method).

See also [28.2.2.5](#) and the **run** option of [29.9](#).

Syntax example:

```
stop_run( )
```



## 25. Messages

### 25.1 Overview

The messaging feature is a centralized and flexible mechanism used to write text messages to various destinations, such as log files or the display. It lets a developer easily insert formatted messages into code and provides the user with powerful and flexible controls to selectively enable or disable groups of messages.

The three most typical uses for messages are:

- a) Summaries—writing summary information at the beginning or end of significant chunks of activity;
- b) Tracing—writing detailed trace messages during the simulation upon interesting events;
- c) Debugging—writing detailed debug messages during the run to help the user or developer debug unexplained behaviors.

Messages are different from plain **out()** and **outf()** calls (see [29.7](#)); they have an optional standard-format prefix and their actions can be disabled or redirected. Messages are also different from **dut\_error()** calls (see [17.2.2](#)); they do not signify failure, increment error counters, or increment warning counters.

### 25.2 The message model

Upon execution, the message action creates a message and sends it to one or more message loggers that have been configured to be a handling logger for this message. Each *handling logger* can be configured to filter messages in various ways, format the enabled messages in various ways (adding the time, name of the unit, etc.), and send them to various destinations (files and the screen).

The message loggers are instantiated by the programmer in the `unit` hierarchy and then configured using constraints (see [25.5](#)) or method calls (see [25.6](#)).

There is a predefined message logger instantiated in every environment under **sys** called **logger**.

### 25.3 message and messagef

<b>Purpose</b>	Create a (formatted) message and send it to a message logger
<b>Category</b>	Action
<b>Syntax</b>	<b>message</b> ([tag], verbosity, exp, ...) [action_block] <b>messagef</b> ([tag], verbosity, format_exp, [exp, ...]) [action_block]
<b>Parameters</b>	<i>tag</i> A constant of type message_tag, either <b>NORMAL</b> or a user-defined tag (see <a href="#">25.3.1</a> ).
	<i>verbosity</i> A constant of type message_verbosity: one of <b>NONE</b> , <b>LOW</b> , <b>MEDIUM</b> , <b>HIGH</b> , or <b>FULL</b> (see <a href="#">25.3.2</a> ).
	<i>exp</i> Value(s) to write.
	<i>action_block</i> A block of actions to perform, the output of which is taken as part of the message output.
	<i>format_exp</i> For <b>messagef()</b> , an <b>outf()</b> -style format string for the output.

When a **message()** or **messagef()** action is executed, the following happens.

- a) If there are no handling loggers for the action (see [25.4.2](#)), then the action is skipped.
- b) If there are handling loggers for the action, the action creates a message (consisting of a list of strings, plus related information) and sends it to all of the handling loggers. Those loggers format the message and send it to the screen and/or log files.
  - 1) For **message()**, the first string in the message is created by appending all of the expressions, like **out()** does.
  - 2) For **messagef()**, the first string is created using the *format-exp*, similar to **outf()**.
    - i) **messagef()** does not automatically add a newline (\n) to the message string. Therefore, if the optional *action-block* requires a newline to be written before it is executed, terminate the *format-exp* using \n.
    - ii) If the fully composed message string—including that portion written by the optional *action-block*—is not terminated by a newline, a newline is appended. **messagef()** also allows appending of the *action-block* output to the **messagef()** header output.
- c) If an *action-block* exists, it gets executed. This typically contains further output-producing actions, calls to reporting methods, etc. The output of all of those is added, as a list of string, to the message.

Message code shall not modify the flow of the simulation in any way. Time-consuming operations in message headers or action blocks are strictly disallowed.

Syntax examples:

```
message(HIGH, "Master ", me, " has received ", the_packet) {
    write the_packet
};
-- Output this message and write the packet, at verbosity HIGH.

message(VR_XBUS_FILE, MEDIUM, "Packet ", num, " sent: ", data)
-- Output this message at verbosity MEDIUM.
-- Use VR_XBUS_FILE as the message-tag.
```

### 25.3.1 Tag

Both **message()** and **messagef()** have an optional first parameter of type `message_tag`, which is initially defined as:

```
type message_tag : [NORMAL]
```

This can be extended, e.g.,

```
extend message_tag : [VR_XBUS_PACKET]
```

If a *tag* is not specified (i.e., the first parameter of **message()** or **messagef()** is a legal value for verbosity), then the value `NORMAL` is prepended. Thus, the following two lines are the same:

```
message(          MEDIUM, "Packet done: ", packet);
message(NORMAL, MEDIUM, "Packet done: ", packet)
```

Message tags are used for associating specific message actions with a message logger (see [25.4](#)).

### 25.3.2 Verbosity

The *verbosity* parameter can be set to **NONE**, **LOW**, **MEDIUM**, **HIGH**, or **FULL** (from lowest to highest). Since a lower verbosity setting means fewer messages are shown, important messages should be assigned a lower *verbosity* parameter value.

[Table 46](#) shows the recommended usage of verbosity. Each level can assume that all lower levels are also writing (thus, there is no need to repeat them).

**Table 46—Verbosity levels**

Level	Recommended use	Examples
NONE	Critical messages (this level cannot be disabled).	"WARNING: Running in reduced mode"
LOW	Messages that happen once per run or once per reset.	"Master M3 was instantiated" "Device D6 got out of reset"
MEDIUM	Short messages that happen once per data item or sequence.	"Packet-@36 was sent to port 7" "A write request to pci bus 2 with address=0xf2223, data=0x48883"
HIGH	More detailed per-data-item information, including: <ul style="list-style-type: none"> <li>— Actual value of the packet</li> <li>— Sub-transaction details</li> </ul>	"Full details for packet-@36: len=5 kind=small ..."
FULL	Anything else, including writing by using specific methods (just to follow the algorithm of that method).	

### 25.3.3 Nested message actions

A message action may be designated as a *leader* by using the **set\_leader()** method (see [25.6.1.7](#)). A *leader* message action is responsible for all message actions inside its lexical scope or in methods called from it. All nested message actions are handled by the leader's loggers (and only by them). The nested messages' text is

formatted sequentially according the leader's rules, as if their output code resides directly inside the leader's lexical scope. This is true even if some of the nested message actions are themselves designated as leaders—the outermost leader overrides the rest.

For sample usage of nested message actions, see [25.7.2](#) and [25.7.3](#).

## 25.4 Message loggers

### 25.4.1 Overview

**message\_logger** is a predefined unit type that is used to manage the output from message actions: filtering it, formatting it, or sending it to one or more destinations. Message loggers are defined programmatically and attached as fields to various units in the unit hierarchy, e.g.,

```
extend vr_xbus_env_u {
    logger : message_logger is instance
}
```

The following can be specified (via methods or constraints) for each message logger:

- Which subset of the message actions the logger examines. By default, each logger uses a verbosity of **NONE** and its tag list is empty. This passes the control to the predefined logger **sys.logger** until the (new) logger is explicitly enabled.
- Which subset of the unit instances the logger examines. By default, this is the tree starting at the unit to which the logger is attached, e.g., each `vr_xbus_env_u.logger` looks at the unit sub-tree under the corresponding `vr_xbus_env_u` unit.
- Which destinations (files or screen) to use for receiving output.
- What format to use.

Loggers, like other units, are generated at elaboration time before the simulation run begins.

### 25.4.2 Handling messages

During execution, messages are processed by loggers in the following manner:

- a) Upon execution of a message action, the set of loggers relevant to that message is determined. The relevant messages are those for which both the following conditions hold.
  - 1) They are associated with the object's origin unit—the unit instance returned by **any\_struct.get\_unit()** (see [7.5.1](#)).
  - 2) The current setting applies to them according to its verbosity, tag, and other criteria (see [25.6.1](#)).
- b) Then, for each of the relevant loggers:
  - 1) **accept\_message()** is called (see [25.6.2](#)); if it returns **FALSE** (due to explicit user refinement), the logger is skipped.
  - 2) **format\_message()** is called (see [25.6.2](#)) to retrieve the final message format.
  - 3) The message in its final format is written to each of the destinations assigned to the logger (the screen and/or any log files).
- c) Each message may be sent to a given destination at most once. If
  - 1) more than one logger is instantiated at the same point in the unit hierarchy;
  - 2) two or more loggers handle the same message and are assigned the same destination; and

- 3) more than one candidate logger still remains after evaluating the other criteria (verbosity, tag, **accept\_message()**, etc.);
- then the message shall go to all distinct destinations defined in the candidate loggers, and the format used when more than one candidate logger specifies the same message shall be *implementation-dependent*.

## 25.5 Configuring message loggers with constraints

Loggers may be configured by using constraints or through a messaging interface (see [25.6](#)). *Constraints* are used during pre-run generation to set the fields of the logger. Then, during **post\_generate()** (see [10.5.3](#)), the logger fields are used to configure the logger. At post-generation, the logger becomes attached to the unit (specifically, the unit computed by *logger-instance.get\_unit()*). The **post\_generate()** method of a logger uses the generated values of the fields to configure the logger via the procedural interface.

[Table 47](#) shows the constrainable fields of the unit *logger*, along with the built-in soft constraints that determine their default values.

**Table 47—Logger constrainable fields**

Field	Description
tags : list of message_tag; keep soft tags == {}	The message tags for selecting the actions for this logger.
verbosity : message_verbosity; keep soft verbosity == NONE	The verbosity for selecting the actions for the logger.
modules : string; keep soft modules == "*"	The modules wild card for selecting the actions for the logger.
string_pattern : string; keep soft string_pattern == "..."	The pattern to match against the string in the message action.
to_file : string; keep soft to_file == ""	The file to which the logger writes (or none if the setting is "").
to_screen : bool; keep soft to_screen == TRUE	When set to TRUE, the logger also writes any output to the screen.

The following considerations also apply:

- Only one file may be associated with a logger via constraints. However, multiple files are allowed when using the messaging procedural interface (see [25.6](#)).
- By default, loggers ignore all tags and have a verbosity of NONE. The one exception is **sys.logger**, where the tag list defaults to {NORMAL} and the verbosity defaults to LOW. Thus, all messages are controlled by **sys.logger**.
- If the verbosity is constrained to a value other than NONE and the tag list is unconstrained, the tag list defaults to {NORMAL}.
- The name of the file written by the logger is appended with the extension `.elog` if the value of the field `to_file` specified by the user does not include an extension. In this context, an *extension* is a period (`.`), followed by one or more characters in the portable filename character set.

## 25.6 Messaging procedural interface (PI)

The following methods of struct type *message\_logger* can be used to set its configuration before or during execution, refine its filtering or formatting rules, and query for details of the currently processed message action. These methods may be used at runtime to override the default (see [25.5](#)) or user-defined settings specified by constraints.

### 25.6.1 Methods for setting configuration

The following methods of struct type *message\_logger* are used to set its configuration.

#### 25.6.1.1 `set_actions`

<b>Purpose</b>	Add, remove, or replace the specified actions for the logger	
<b>Category</b>	Method	
<b>Syntax</b>	<b><code>set_actions</code></b> ( <i>verbosity</i> , <i>tags</i> , <i>modules</i> , <i>text</i> , <i>op</i> )	
<b>Parameters</b>	<i>verbosity</i>	A constant of type <i>message_verbosity</i> : one of <b>NONE</b> , <b>LOW</b> , <b>MEDIUM</b> , <b>HIGH</b> , or <b>FULL</b> (see <a href="#">25.3.2</a> ). This matches the message actions whose verbosity is between <b>LOW</b> and <i>verbosity</i> (when <i>op</i> = <b>add</b> or <b>replace</b> ) or between <i>verbosity</i> and <b>FULL</b> (when <i>op</i> = <b>remove</b> ).
	<i>tags</i>	Matches the message actions whose tags (see <a href="#">25.3.1</a> ) are those specified by the list (of type <i>message_tag</i> ).
	<i>modules</i>	Matches the message actions residing in the module(s) that match this string expression.
	<i>text</i>	Matches the message actions, of which one of the parameters is a literal string that matches the string expression.
	<i>op</i>	One of <b>add</b> , <b>replace</b> , or <b>remove</b> . This determines whether the message actions matched by previous parameters are added to the logger, removed from it, or used to replace the currently assigned action.

This adds, removes, or replaces the specified actions for the logger.

Syntax example:

```

extend vr_xbus_env_u {
  run() is also {
    logger.set_actions(FULL, {NORMAL}, "*", "...", replace)
  }
}

```



### 25.6.1.2 set\_unit

<b>Purpose</b>	Set the unit tree under a specified unit to be on or off for the message logger	
<b>Category</b>	Method	
<b>Syntax</b>	<b>set_unit</b> ( <i>root</i> , <i>status</i> )	
<b>Parameters</b>	<i>root</i>	The root unit of the unit tree to add to or remove from the set of unit instances associated with the logger.
	<i>status</i>	One of <b>on</b> or <b>off</b> ; this determines whether the units are added or removed.

This adds (**on**) or removes (**off**) units from the logger.

Syntax example:

```

extend vr_xbus_env_u {
    run() is also {
        logger.set_unit(testbench_01, off)
    }
}

```

### 25.6.1.3 set\_format

<b>Purpose</b>	Set the format for messages associated with the message logger	
<b>Category</b>	Method	
<b>Syntax</b>	<b>set_format</b> ( <i>format</i> )	
<b>Parameters</b>	<i>format</i>	A value of type <b>message_format</b> . The predefined choices are <b>short</b> , <b>long</b> , or <b>none</b> ; this can be extended by the user. <b>none</b> specifies no additions to the bare message text. Any styles implied by the other formats are implementation-dependent.

This sets the format mode for any messages associated with the logger.

Syntax example:

```

extend vr_xbus_env_u {
    run() is also {
        logger.set_format(none)
    }
}

```

#### 25.6.1.4 set\_flush\_frequency

<b>Purpose</b>	Set the message flushing frequency of each message output file associated with the logger
<b>Category</b>	Method
<b>Syntax</b>	<b>set_flush_frequency</b> ( <i>num</i> )
<b>Parameters</b>	<i>num</i> The maximum number of message actions executed before their content is flushed out.

This sets the maximum number of message actions allowed before flushing the message queue for the logger.

Syntax example:

```

extend vr_xbus_env_u {
    run() is also {
        logger.set_flush_frequency(15)
    }
}

```

#### 25.6.1.5 set\_file

<b>Purpose</b>	Add a file to or remove a file from the specified logger
<b>Category</b>	Method
<b>Syntax</b>	<b>set_file</b> ( <i>fname</i> , <i>status</i> )
<b>Parameters</b>	<i>fname</i> The log file to add or remove.
	<i>status</i> One of <b>on</b> or <b>off</b> ; this determines whether the files are added or removed.

This adds a file to the specified logger (or if **off** is specified, removes it from the logger). Each logger can write to any number of files, including writing to the screen (see [25.6.1.6](#)).

Syntax example:

```

extend vr_xbus_env_u {
    run() is also {
        var my_file : file;

        my_file = files.open("my.log", "w", "Log file");
        logger.set_file(my_file, on)
    }
}

```

### 25.6.1.6 set\_screen

<b>Purpose</b>	Enable or disable the logger from writing to the screen
<b>Category</b>	Method
<b>Syntax</b>	<b>set_screen</b> ( <i>status</i> )
<b>Parameters</b>	<i>status</i> Either <b>on</b> (writing to screen is enabled) or <b>off</b> (writing to screen is disabled).

This enables or disables the logger from writing to the screen. (Any messages sent to the screen are directed the same way as text from the **out()** pseudo-routine and the print action.)

Syntax example:

```

extend vr_xbus_env_u {
  run() is also {
    if logger.to_file != "" then {
      logger.set_screen(off)
    }
  }
}

```

### 25.6.1.7 set\_leader

<b>Purpose</b>	Set the specified messages' actions as leaders or non-leaders.
<b>Category</b>	Method
<b>Syntax</b>	<b>set_leader</b> ( <i>verbosity, tags, modules, text, status</i> )
<b>Parameters</b>	<i>verbosity</i> A constant of type <code>message_verbosity</code> : one of <b>NONE</b> , <b>LOW</b> , <b>MEDIUM</b> , <b>HIGH</b> , or <b>FULL</b> (see <a href="#">25.3.2</a> ). This matches the message actions whose verbosity is between <b>LOW</b> and <i>verbosity</i> (when <b>set_actions</b> <i>op</i> = <b>add</b> or <b>replace</b> ) or between <i>verbosity</i> and <b>FULL</b> (when <b>set_actions</b> <i>op</i> = <b>remove</b> ). See also <a href="#">25.6.1.1</a> .
	<i>tags</i> Matches the message actions whose tags (see <a href="#">25.3.1</a> ) are those specified by the list (of type <code>message_tag</code> ).
	<i>modules</i> Matches the message actions residing in the module(s) that match this string expression.
	<i>text</i> Matches the message actions, of which one of the parameters is a literal string that matches the string expression.
	<i>status</i> One of <b>on</b> or <b>off</b> ; this determines whether to set the specified messages' actions as leaders ( <b>on</b> ) or non-leaders ( <b>off</b> ).

This sets the specified actions to be leaders (or if **off** is specified, to cease from being leaders); see also [25.3.3](#).

Syntax example:

```

extend vr_xbus_env_u {
  run() is also {

```

```

    logger.set_leader(LOW, {NORMAL}, "*", "...", on)
  }
}

```

### 25.6.1.8 ignore\_tags

<b>Purpose</b>	Ignore the specified tags
<b>Category</b>	Method
<b>Syntax</b>	<b>ignore_tags</b> ( <i>tags</i> )
<b>Parameters</b>	<i>tags</i> Ignores the message actions whose tags are those specified by the list (of type <code>message_tag</code> ).

This causes the logger to ignore messages of this tag type.

Syntax example:

```

extend message_tag : [VR_XBUS_PACKET];

extend vr_xbus_env_u {
  run() is also {
    logger.ignore_tags({VR_XBUS_PACKET})
  }
}

```

### 25.6.2 Hook methods for refining message handling

The predefined methods shown in this subclause are automatically called during message handling. They can also be extended to override the default behavior.

#### 25.6.2.1 accept\_message

<b>Purpose</b>	Check if the current message is enabled
<b>Category</b>	Method
<b>Syntax</b>	<b>accept_message()</b> :bool
<b>Parameters</b>	None
<b>Return value</b>	A Boolean value

This returns `TRUE` (the default) if the current message is enabled. When this method returns `FALSE` (see the following example), the current message is ignored by the logger and the logger destination is blocked for the current message.

Syntax example:

```

extend message_logger {
  accept_message(): bool is only {
    return get_tag() == normal
  }
}

```

```
}
```

### 25.6.2.2 format\_message

<b>Purpose</b>	Format a message
<b>Category</b>	Method
<b>Syntax</b>	<b>format_message()</b> :list of string
<b>Parameters</b>	None
<b>Return value</b>	The message

This returns the *list of string*, which will be sent as-is to the file or screen.

Syntax example:

```

extend message_logger {
  format_message(): list of string is only {
    var msg_list := get_message();
    ...
  }
}

```

### 25.6.3 Query methods for getting message information

The methods shown in this subclause are available for use within the **accept\_message()** (see [25.6.2.1](#)) and **format\_message()** methods (see [25.6.2.2](#)). Query methods return information about the message action that was just executed.

#### 25.6.3.1 get\_format

<b>Purpose</b>	Get a message's format
<b>Category</b>	Method
<b>Syntax</b>	<b>get_format()</b> :message_format
<b>Parameters</b>	None
<b>Return value</b>	The current <b>message_format</b> value for the logger. The built-in choices for <b>message_format</b> are <b>short</b> (the default), <b>long</b> , or <b>none</b> ; the user is allowed to extend the type.

This returns the message format of the current logger.

### 25.6.3.2 get\_message

<b>Purpose</b>	Get the current raw message
<b>Category</b>	Method
<b>Syntax</b>	<b>get_message()</b> :list of string
<b>Parameters</b>	None
<b>Return value</b>	The message

This returns the current raw message as produced by the message action.

### 25.6.3.3 get\_message\_action\_id

<b>Purpose</b>	Get the message ID
<b>Category</b>	Method
<b>Syntax</b>	<b>get_message_action_id()</b> :int
<b>Parameters</b>	None
<b>Return value</b>	The message ID

This returns a unique number identifying the message action.

### 25.6.3.4 get\_tag

<b>Purpose</b>	Get a message's tag
<b>Category</b>	Method
<b>Syntax</b>	<b>get_tag()</b> :message_tag
<b>Parameters</b>	None
<b>Return value</b>	A message-tag (see <a href="#">25.3.1</a> )

This returns the tag of the message action.

### 25.6.3.5 `get_verbosity`

<b>Purpose</b>	Get a message's verbosity
<b>Category</b>	Method
<b>Syntax</b>	<b><code>get_verbosity()</code></b> :message_verbosity
<b>Parameters</b>	None
<b>Return value</b>	The message's verbosity (see <a href="#">25.3.2</a> )

This returns the verbosity of the message action.

### 25.6.3.6 `source_location`

<b>Purpose</b>	Get the message's location
<b>Category</b>	Method
<b>Syntax</b>	<b><code>source_location()</code></b> :string
<b>Parameters</b>	None
<b>Return value</b>	The source location

This returns the source location where the message occurred, e.g., "At line 12 in @foo".

### 25.6.3.7 `source_method_name`

<b>Purpose</b>	Get the method name for the current message
<b>Category</b>	Method
<b>Syntax</b>	<b><code>source_method_name()</code></b> :string
<b>Parameters</b>	None
<b>Return value</b>	The method's name

This returns the name of the method where the message occurred, e.g., "foo".

### 25.6.3.8 source\_struct

<b>Purpose</b>	Get the message's source struct
<b>Category</b>	Method
<b>Syntax</b>	<b>source_struct()</b> :any_struct
<b>Parameters</b>	None
<b>Return value</b>	A struct

This returns the struct where the message occurred.

### 25.6.3.9 source\_struct\_name

<b>Purpose</b>	Get a message's struct type-name
<b>Category</b>	Method
<b>Syntax</b>	<b>source_struct_name()</b> :string
<b>Parameters</b>	None
<b>Return value</b>	The struct's type-name

This returns the name of the struct type where the message occurred, e.g., "packet".

## 25.7 Examples

### 25.7.1 Example 1

Add some message actions, and put a logger in the unit.

```

unit my_dsp {
  foo() is {
    ...
    message(LOW, "Starting transmission");
    -- LOW verbosity means this is an important message that
    -- will be shown even when verbosity is set to 'LOW'
    ...
    message(MEDIUM, "Sending packet ", pkt);
    -- MEDIUM verbosity means this is a less important message
    ...
  };

  logger: message_logger is instance
  -- Instantiate a message logger for this unit. When activated,
  -- it will handle all message actions executing in this unit
  -- or in any unit or struct under it.
}

```



Configure the logger via constraints.

```

extend sys {
    dsp : my_dsp is instance;
    keep dsp.logger.tags == {NORMAL};
    keep dsp.logger.verbosity == LOW;
    -- This logger will only look at important messages
    keep dsp.logger.to_file == "dsp_results.elog"
    -- Send it also to a file (it goes to the screen by default)
}

```

### 25.7.2 Example 2

Confirm the writing order, assuming the following code:

```

message(NONE, "I'm A0") {
    out("A1");
    message(NONE, "I'm B0") {
        out("B1");
        message(NONE, "I'm C0") {
            out("C1")
        };
        out("B2")
    };
    out("A2")
}

```

If there is no leader message, then the output is written in the order that the message blocks finish:

```

I'm C0
C1
I'm B0
B1
B2
I'm A0
A1
A2

```

With "I'm A0" as the leader message, the order would be:

```

I'm A0
A1
I'm B0
B1
I'm C0
C1
B2
A2

```

### 25.7.3 Example 3

This example illustrates the leader message concept. The message action "goo: I'm a message from A" is nested within the lexical scope of the message action "foo I'm U2". If neither of these messages is designated a leader (see [25.3.3](#) and [25.6.1.7](#)), "goo: I'm a message from A" is handled by logger `sys.u1.l1` and "foo I'm U2" by `sys.u2.l2`. However, if the outermost message action "foo I'm U2" is a leader, both messages are handled by `sys.u2.l2`, irrespective of the leader designation of the inner message action "goo: I'm a message from A".

```

extend sys {

```

```

    u1 : U1 is instance;
    u2 : U2 is instance;

    run() is also {
        u2.foo()
    }
};

unit U1 {
    l1 : message_logger is instance;
    keep l1.verbosity == FULL;
    a : A
};

unit U2 {
    l2 : message_logger is instance;
    keep l2.verbosity == FULL;

    foo() is {
        message(NONE, "foo: I'm U2") {
            out("foo: before");
            sys.u1.a.goo();
            out("foo: after")
        }
    }
};

struct A {
    goo() is {
        message(NONE, "goo: I'm a message from A")
    }
}

```

If there is no leader message, then the output would be:

```

goo: I'm a message from A    -- handled by sys.u1.l1
foo: I'm U2                 -- handled by sys.u2.l2
foo: before                 -- handled by sys.u2.l2
foo: after                  -- handled by sys.u2.l2

```

To change the leader message, the following code can be added to the example:

```

extend sys {
    run() is first {
        u2.l2.set_leader(NONE, {NORMAL}, "*", "foo: I'm U2", on)
    }
}

```

With "foo: I'm U2" as the leader message, the order would be:

```

foo: I'm U2                 -- handled by sys.u2.l2
foo: before                 -- handled by sys.u2.l2
goo: I'm a message from A   -- handled by sys.u2.l2
foo: after                  -- handled by sys.u2.l2

```

## 26. Sequences

### 26.1 Overview

*Sequences* provide a uniform way to define streams of data items and compose them into verification scenarios of growing complexity. *Sequence items* are typically driven into a DUT, but the details of the interaction with the DUT are decoupled from the data stream creation and hidden behind a standard interface.

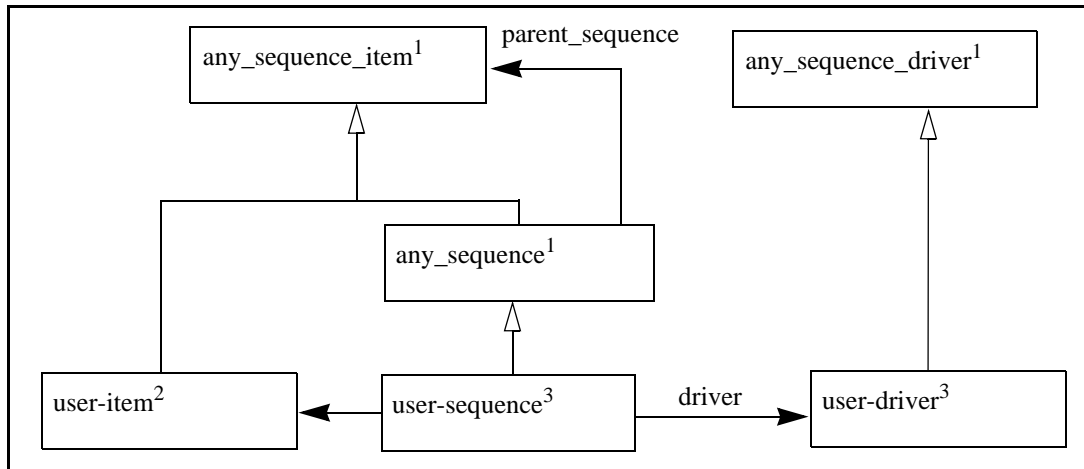
For defining sequences, it is also necessary to define standard interfacing entities between the sequence and the DUT. Therefore, the *sequence model* has three main entities.

- a) *Item*—A struct that represents the main input to the DUT (e.g., packet, transaction, or instruction).
- b) *Sequence*—A struct that defines a stream of items representing a high-level stimuli scenario. This is done by generating items one after the other, according to some specific rules. The sequence struct has a set of predefined fields and methods. The user can also extend this struct.
- c) *Sequence driver*—A unit that serves as the mediator between the sequences and a verification environment. The generated items are passed from the sequence to the sequence driver; the sequence driver acts upon them one-by-one, typically passing them to some kind of bus functional model (BFM).
  - 1) The sequence driver and the BFM work as a pair, where the sequence driver serves as the interface upwards towards the sequences so the sequences can always see a standard interface to the DUT. The BFM serves as the interface downwards to the DUT, allowing the user to write sequences as appropriate. The importance of maintaining the separation between the sequence driver and the BFM becomes clear when implementing virtual sequences (see [26.1.2](#)).
  - 2) To complete the picture:
    - i) A TCM does the actual driving of items into a specific DUT channel.
    - ii) The TCM resides in a BFM unit.
    - iii) For the purpose of driving data into the DUT, the sequence driver interacts only with the BFM.

From the point of view of a programming language, a sequence is best described as the instantiation of a *design pattern*. The same pattern needs to be instantiated separately for every type of data item that constitutes verification scenarios for the DUT. The basic implementation of the pattern is given to a user-defined sequence through *like* inheritance, and through type-parameterized code (a template), where the parameter is the data-item type.

#### 26.1.1 Object model

The actual sequence object model consists of six struct types and their relationships, shown as a UML diagram in [Figure 15](#).



<sup>1</sup> Built-in struct type.

<sup>2</sup> Declared by user.

<sup>3</sup> Declared by a *sequence* statement.

**Figure 15—UML diagram of sequence object model types**

For a UML diagram of this type, the styles of arrow heads *are significant*. There are two kinds of relationships depicted in [Figure 15](#), the directional association (the horizontal connectors from user-sequence) and inheritance. The inheritance relationship in UML is depicted by an open, triangular arrowhead pointing to the base class (e.g., user-driver to *any\_sequence\_driver*). One or more lines proceed from the base of the arrowhead connecting it to the derived classes (e.g., for *any\_sequence\_item*).

### 26.1.2 Virtual sequences

BFM sequences are tightly connected to their own type and items; BFM sequences cannot **do** sequences created by other sequence statements. *Virtual sequences*, however, are not tightly connected and can **do** sequences of other types (but not items). Hence, they can be used to drive more than one agent, model a generic driver, or to synchronize and dispatch BFM sequences to several BFM drivers.

A virtual sequence is driven by a *virtual sequence driver*, which typically has references to the individual BFM sequence drivers; however, a virtual sequence driver is not connected to a specific BFM. Therefore, it lacks the logic and functionality of a BFM driver, e.g., a virtual sequence driver does not schedule items—it only drives sequences. As part of a driver’s functionality is aimed at controlling and manipulating the scheduling of items, any method that controls this functionality cannot be called for a virtual sequence. For the full list of driver interface methods that cannot be used for virtual sequences, see [Table 50](#).

**NOTE**—To activate single items from a virtual sequence, use the `SIMPLE` sequence. To pass any parameters to the item, define the parameters as fields in the `SIMPLE` sequence and propagate them by constraining the item.

## 26.2 Sequence statement

<b>Purpose</b>	Declare and partially define the sequence struct, the sequence driver unit, and the sequence kind enumerated type. For BFM sequences, extend the item struct for collaboration in the sequence pattern
<b>Category</b>	Statement
<b>Syntax</b>	<b>sequence</b> <i>sequence_type_name</i> [ <b>using</b> <i>sequence_option</i> , ...]
<b>Parameters</b>	<i>sequence_type_name</i> The name of the new sequence.
	<i>sequence_option</i> <i>sequence_option</i> is one of the following: <ul style="list-style-type: none"> <li>a) <b>item</b> = <i>item_type</i>—the item to use in the sequence. This struct type shall already be defined and inherit from <b>any_sequence_item</b>. The item struct is extended by the sequence statement; the sequence is a <i>BFM sequence</i>. If this option is not used, this sequence is a <i>virtual sequence</i>.</li> <li>b) <b>created_kind</b> = <i>kind_type_name</i>—the associated kind enumerated type to create; the default is <i>sequence_type_name_kind</i>.</li> <li>c) <b>created_driver</b> = <i>driver_type_name</i>—the associated sequence driver to create; the default is <i>sequence_type_name_driver</i>.</li> <li>d) <b>sequence_type</b> = <i>base_sequence_type</i>—the sequence struct used for inheritance. This struct shall inherit from <b>any_sequence</b>; the default is <i>any_sequence</i>.</li> <li>e) <b>sequence_driver_type</b> = <i>base_sequence_driver_type</i>—the sequence driver unit used for inheritance. This unit shall inherit from <b>any_sequence_driver</b>; the default is <i>any_sequence_driver</i>.</li> </ul>

The **sequence** statement creates a new sequence struct, which inherits from the predefined **any\_sequence**, which in turn inherits from **any\_sequence\_item**. It also creates a new sequence driver unit, which inherits from **any\_sequence\_driver**. Finally, it extends the (user-defined) item struct. For more details on all the resulting struct type members, see [26.4](#). For further details about the resulting predefined sequence kinds, see [26.2.1](#).

Syntax example:

```
sequence ex_atm_sequence using item=ex_atm_cell
```

### 26.2.1 Predefined sequence kinds

The enumerated type of the sequence kind field contains the predefined values MAIN, RANDOM, and SIMPLE for BFM sequences and only MAIN and RANDOM for virtual sequences.

#### 26.2.1.1 MAIN

This sequence subtype is instantiated directly under the sequence driver and is started by default. It is used as the root of the whole sequence tree. See also the *gen\_and\_start\_main* driver field, which controls automatic start of MAIN sequence, in [Table 50](#).

```
define MAX_RANDOM_COUNT 10;

extend MAIN sequence_name {
    count    : uint;
```

```

!sequence : sequence_name;

keep soft count > 0;
keep max_random_count <= MAX_RANDOM_COUNT;
keep soft count <= MAX_RANDOM_COUNT;
keep sequence.kind not in [RANDOM, MAIN];

body() @driver.clock is only {
    for i from 1 to count do {
        do sequence
    }
}

```

### 26.2.1.2 SIMPLE

This sequence subtype generates and executes a single item for BFM sequences.

```

extend SIMPLE sequence_name {
    !seq_item: item;

    body() @driver.clock is only {
        do seq_item
    }
}

```

### 26.2.1.3 RANDOM

This sequence subtype is used for creating random scenarios based on SIMPLE and user-defined sequence subtypes.

```

extend RANDOM sequence_name {
    count : uint;
    !sequence : sequence_name;

    keep all of {
        soft count > 0;
        max_random_count <= MAX_RANDOM_COUNT;
        soft count <= MAX_RANDOM_COUNT;
        sequence.kind not in [RANDOM, MAIN];
        depth_from_driver >= driver.max_random_depth =>
            sequence.kind == SIMPLE
    };

    body() @driver.clock is only {
        for i from 1 to count do {
            do sequence
        }
    }
}

```

## 26.2.2 Examples

### Example 1

This example defines a BFM sequence for ATM cells.

```
sequence ex_atm_sequence using item=ex_atm_cell
```

It assumes the `ex_atm_cell` struct already exists and establishes the following definitions:

```
struct      ex_atm_sequence (inherits from any_sequence)

type        ex_atm_sequence_kind (predefined items MAIN, RANDOM, and
SIMPLE).

unit        ex_atm_sequence_driver (inherits from any_sequence_driver)
```

### Example 2

This example defines a virtual sequence for an SOC environment; it defines the `soc_sequence` struct, the `soc_sequence_kind` type, and the `soc_sequence_driver` unit.

```
sequence soc_sequence
```

## 26.3 do sequence action

<b>Purpose</b>	Generate and drive an item or subsequence declared as fields in the enclosing sequence	
<b>Category</b>	Action	
<b>Syntax</b>	<b>do</b> <i>field_name</i> [ <b>keeping</b> { <i>constraint</i> [ <b>;</b> <i>constraint</i> ] ...}]	
<b>Parameters</b>	<i>field_name</i>	A field in the current struct. It shall be an ungeneratable field, as indicated by a leading exclamation mark (!), and also needs to be a basic item or a sequence.
	<i>constraint</i>	Any generation constraints on <i>field_name</i> .

The **do** action performs the following steps (see also [Table 49](#) and [Table 50](#) and [Figure 17](#), [Figure 18](#), [Figure 19](#), and [Figure 20](#)).

- a) On a subsequence:
  - 1) Generates the field, considering the constraints, if any.
  - 2) Calls its **body()** TCM.

The **do** action finishes when the subsequence **body()** returns.
- b) On an item:
  - 1) Waits until the driver is ready to perform the **do** action.
  - 2) Generates the field, considering the constraints, if any.

The item is returned by **get\_next\_item()**.

The **do** action finishes when *driver.item\_done* is emitted.

The following considerations also apply:

- **do** is a time-consuming, atomic action—as observed from the thread executing the **do** action—that activates an item or sequence.
- The **do** action can only be activated inside sequences.
- BFM sequences cannot **do** sequences created by other sequence statements.
- When **do**-ing an item, emit the event *driver.item\_done* to let the sequence complete the **do** action and inform the driver the item was processed (typically, after the transmission of the item via the BFM.) Otherwise, the sequence cannot continue and the driver cannot drive more items.
- For items, waiting for the sequence driver to be ready is performed before generation to ensure generation is done as close to the actual driving as possible. Thus, if the constraints depend on the current status of the DUT/environment, that status is as accurate as possible.
- The sequence driver decides when the item is ready by managing a FIFO that also considers any **grab/ungrab** actions done by the various sequences and the **is\_relevant()** sequence value. If no grab is done and **is\_relevant()** returns TRUE for all sequences, the order of doing the items is determined by the order of the **do** actions in the various sequences that refer to the sequence driver, regardless of their depth or origin. Sequences and items can also be done in parallel, using the **all of** and **first of** actions. (See also **grab()** and **is\_relevant()** in [Table 49](#).)

Syntax example:

```

extend FOO ex_atm_sequence {
    // Parameters
    i    : int;
    b    : bool;

    // Items/subsequences
    !cell : ex_atm_cell;
    !seq  : BAR ex_atm_sequence;

    // The body() method
    body() @driver.clock is {
        do cell keeping {.len == 4};
        do cell;

        for i = 1 to 20 do {
            do cell keeping {.address == i}
        };

        do seq keeping {.f == 2}
    }
}

```

## 26.4 Sequence struct types and members

This subclause describes the entities the sequence statement extends (the sequence-item structs) or creates (the sequence and sequence-driver structs). For the RO/RW columns in [Table 48](#), [Table 49](#), and [Table 50](#), RO designates a read-only member (which can only be read or invoked) and RW designates a read/write member (which can also be set, constrained, or implemented).

### 26.4.1 Sequence item API

The **sequence** statement does not create the item struct, but it extends it. The user needs to create the item struct, which inherits from **any\_sequence\_item**. The main members that are added to the item struct are shown in [Table 48](#).



**Table 48—Sequence item struct members**

Struct member	Description	RO/RW
<b>do_location()</b> : string	Returns a string describing the source location of the <b>do</b> action that generated the current item. Not relevant for <b>sequence_items</b> not created via the <b>do</b> action.	RO
<b>driver</b> : <i>driver_type</i>	Driver for the item, soft-constrained to its parent sequence's driver.	RW
<b>get_depth()</b> : int	Depth from the sequence driver, valid from pre-generation.	RO
<b>get_driver()</b> : <i>driver_type</i>	Returns the driver for an item.	RO
<b>nice_string()</b> : <i>string</i> is empty	A short string representing the sequence item that may be used to aid debug tracing.	RW
<b>!parent_sequence</b> : <i>any_sequence</i>	Back-pointer to the sequence in which an item was created. Assigned automatically in the <b>pre_generate()</b> of the item. <sup>a</sup>	RO

<sup>a</sup>Never use **is only** on the **pre\_generate()** or **post\_generate()** of items or sequences. The field **parent\_sequence** is assigned in the **pre\_generate()** of **any\_sequence\_item**.

#### 26.4.2 Sequence API

The **sequence** statement creates a new sequence struct, which inherits from the predefined **any\_sequence**, which in turn inherits from **any\_sequence\_item**. The main members of the created sequence struct are shown in [Table 49](#).

**Table 49—Sequence struct members**

Struct member <sup>a</sup>	Description	RO/RW
<b>body()</b> @ <i>driver.clock</i> is empty	Main method called by <b>do</b> of parent sequence after it generates the current sequence.	RW
<b>do_location()</b> : string	Returns a string describing the source location of the <b>do</b> action that generated the current item. Not relevant for <b>sequence_items</b> not created via the <b>do</b> action.	RO
<b>driver</b> : <i>driver_type</i>	Driver for the sequence, soft-constrained to its parent sequence's driver.	RW
<b>event</b> ended	Emitted immediately after <b>body()</b> is finished. As part of the service API, this event shall not be explicitly emitted by the sequence user.	RO
<b>event</b> started	Emitted just before <b>body()</b> is called. As part of the service API, this event shall not be explicitly emitted by the sequence user.	RO
<b>get_depth()</b> : int	Depth from the sequence driver, valid from pre-generation.	RO
<b>get_driver()</b> : <i>driver_type</i>	Returns the driver for the sequence.	RO
<b>get_index()</b> : int	Starts at zero (0) and gets incremented after every <b>do</b> .	RO
<b>grab</b> ( <i>driver</i> : <i>any_sequence_driver</i> ) @ <b>sys.any</b>	Grabs the sequence driver for exclusive access and returns when exclusive access has been granted.	RO
<b>is_blocked()</b> : bool	Returns TRUE if the sequence is blocked by another sequence that grabbed the sequence driver.	RO

**Table 49—Sequence struct members (continued)**

Struct member <sup>a</sup>	Description	RO/RW
<b>is_relevant()</b> : bool	Returns TRUE (the default) if the sequence is currently allowed to do items. Use this to respond to the changing conditions of the simulation. See also <a href="#">Figure 18</a> and <a href="#">Figure 19</a> .	RW
<i>kind</i> : kind_type	The kind field that determines which sequence it is (within its when family). <i>kind</i> is declared <b>const</b> and hence cannot change value after the sequence struct generation (see <a href="#">6.8</a> ).	RW
<b>mid_do</b> ( <i>s</i> : any_sequence_item) is empty	A hook method called in the middle of <b>do</b> , just after item <i>s</i> is generated and before it is executed by calling the <b>body()</b> TCM.	RW
<b>nice_string()</b> : <i>string</i> is empty	A short string representing the sequence item that may be used to aid debug tracing.	RW
<b>!parent_sequence</b> : any_sequence	Back-pointer to the sequence in which this sequence was created. Assigned automatically in the <b>pre_generate()</b> of the sequence if such a parent exists. <sup>b</sup>	RO
<b>post_body()</b> <b>@sys.any</b> is empty	A hook method called after <b>body()</b> when sequence is started using the <b>start_sequence()</b> method.	RW
<b>post_do</b> ( <i>s</i> : any_sequence_item) is empty	A hook method called at the end of a <b>do</b> , just after the execution of <i>s.body()</i> .	RW
<b>post_do_tcm</b> ( <i>s</i> : any_sequence_item) <b>@sys.any</b> is empty	A hook TCM called after <b>post_do()</b> that extends the life of a <b>do</b> after its <i>item_done</i> event is emitted. The sequence driver, freed by the <i>item_done</i> event, no longer manages the current item so it can now handle other items.	RW
<b>pre_body()</b> <b>@sys.any</b> is empty	A hook method called before <b>body()</b> when a sequence is started using the <b>start_sequence()</b> method.	RW
<b>pre_do</b> ( <i>is_item</i> : bool) <b>@sys.any</b> is empty	A hook TCM called at the start of a <b>do</b> performed by the sequence. <i>is_item</i> specifies whether the context is <b>do</b> -ing an item or a sequence.	RW
<b>start_sequence()</b>	Starts sequence activity by starting <b>body()</b> . Call this method instead of starting <b>body()</b> directly.	RO
<b>stop()</b>	Terminates (on the cycle boundary) the execution of the thread of that sequence's body TCM if it is executing at that time. It consequently terminates other enclosed sequences. The <b>post_body()</b> of a stopped sequence is not called nor is the <b>post_do()</b> of the <b>do</b> -ing sequence.	RO
<b>ungrab</b> ( <i>driver</i> : any_sequence_driver)	Releases the grab on a sequence driver and returns immediately.	RO

<sup>a</sup>Some of these methods are inherited from the **any\_sequence\_item** interface shown in [Table 48](#).

<sup>b</sup>Never use **is only** on the **pre\_generate()** or **post\_generate()** of items or sequences.

### 26.4.3 Sequence driver API

The **sequence** statement creates a new sequence driver unit, which inherits from **any\_sequence\_driver**. The main members of the created driver unit are shown in [Table 50](#).

**Table 50—Sequence driver unit members**

Struct member	Description	RO/RW	BFM only <sup>a</sup>
<i>bfm_interaction_mode</i> : bfm_interaction_mode_t	Specifies the way the driver and the BFM interact with each other. Possible options are PULL_MODE (the default) and PUSH_MODE. It can be constrained.	RW	Yes
<b>branch_terminated()</b>	Enables resumption of normal operation in the current cycle after the enclosing <b>first of</b> completes.	RO	No
<b>check_is_relevant()</b>	Forces a driver to recheck the relevance (value of <b>is_relevant()</b> ) for each sequence that has items in the driver's item queue. Useful when something has changed in the BFM that affects the relevance of some sequences.	RO	Yes
<b>current_grabber()</b> : any_sequence	Indicates which sequence (if any) has exclusive control over a sequence driver.	RO	Yes
<b>delay_clock()</b> @sys.any	Emits the driver's clock with an inter-cycle delay to allow the environment to execute before the sequences. For example, a BFM may need to export its status before its sequences are evaluated so they can use the new status to generate updated data. This TCM replaces the regular clock connection so the clock event should not be emitted.	RW	No
<b>event</b> clock	The main clock. The user needs to tie this to a TE during the sequence driver hook-up.	RW	No
<b>event</b> item_done	Synchronization event for the <b>do</b> action in PULL_MODE. Emit this event to complete the <b>do</b> item and let the driver get more items using <b>get_next_item()</b> .	RW	Yes
<i>gen_and_start_main</i> : bool	Enables or disables automatic generation and launch of the MAIN sequence upon <b>run()</b> . The default is TRUE (the MAIN sequence is generated and started).	RW	No
<b>get_current_item()</b> : any_sequence_item	Returns the item currently being sent. (NULL if the BFM is currently idle.)	RO	Yes
<b>get_next_item()</b> : item_type @clock	Call this TCM in PULL_MODE to receive the next item from the BFM driver. This TCM is blocked until there is an item to <b>do</b> in the driver.	RO	Yes
<b>get_num_items_sent()</b> : int	Returns the count of items sent (excluding current_item, if any).	RO	Yes
<b>get_sub_drivers()</b> : list of any_sequence_driver is empty	For virtual sequence drivers, the writer of the specific sequence driver needs to fill in this method. It needs to return the list of subdrivers of the sequence driver. For a BFM sequence driver, this would be unchanged, i.e., it returns an empty list.	RW	Virtual only <sup>b</sup>
<b>is_grabbed()</b> : bool	Indicates the grab status of the sequence driver.	RO	Yes
<b>last(index)</b> : any_sequence_item	Enables access to previously sent items in the sequence driver.	RO	Yes
<i>max_random_count</i> : int	Sets the maximum number of subsequences in a RANDOM or MAIN sequence, e.g., keep soft max_random_count == MAX_RANDOM_COUNT; // Defined to be 10	RW	No

**Table 50—Sequence driver unit members (continued)**

Struct member	Description	RO/RW	BFM only <sup>a</sup>
<i>max_random_depth</i> : int	Sets the maximum depth inside a RANDOM sequence. (Beyond that depth, RANDOM creates only SIMPLE sequences.), e.g., <pre>keep soft max_random_depth == MAX_RANDOM_DEPTH; // Defined to be 4</pre>	RW	No
<i>num_of_last_items</i> : int	Sets the length of the history of previously sent items. The default is 1.	RW	Yes
<b>regenerate_data()</b> is empty	Hook method to regenerate driver's data upon <b>rerun()</b> . Never use <b>is only</b> on the <b>run()</b> or <b>rerun()</b> of drivers. Some important initializations are performed in those methods.	RW	No
<b>send_to_bfm</b> (seq_item: item_name) @clock is empty	When working in PUSH_MODE, sends the item to the corresponding BFM. The user needs to implement this as part of achieving the hookup.	RW	Yes
<b>try_next_item</b> (): item_type @clock	Call this TCM in PULL_MODE when the BFM has to receive an item or perform some default behavior. Unlike <b>get_next_item()</b> , when there is no available item waiting to be done in the current cycle, this TCM returns NULL. <b>try_next_item()</b> returns in the same cycle if there is no pending <b>do</b> action. However, if a <b>do</b> action started execution, then <b>try_next_item()</b> might take longer than a cycle, e.g., if the <b>pre_do()</b> TCM has been extended to take longer than a cycle.	RO	Yes
<b>wait_for_sequences</b> () @sys.any	Call this TCM to delay the return of <b>try_next_item()</b> and let sequences create items. It may also be used to effect a thread context switch to force a sequence producer to be evaluated before a consumer. The body of this TCM may be replaced ("wait_for_sequences() @sys.any is only { . . . }") to implement an alternate synchronization algorithm. However, this also influences the behavior of <b>try_next_item()</b> .	RW	Yes

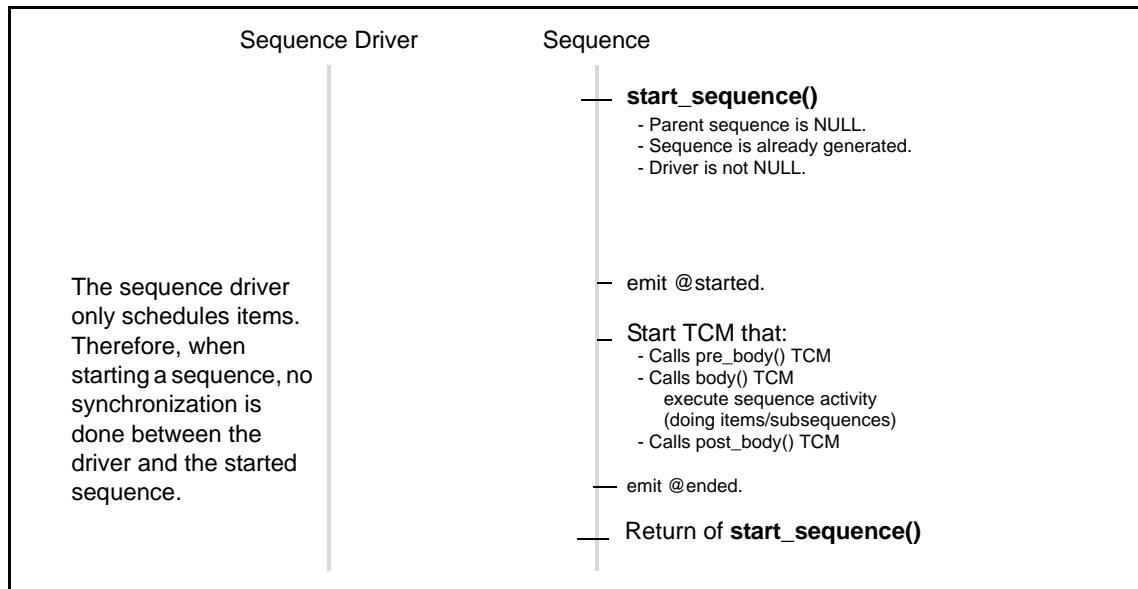
<sup>a</sup>Some of the driver members are relevant only for BFM drivers.<sup>b</sup>This method can only be used for virtual sequences.

## 26.5 BFM-driver-sequence flow diagrams

This subclause shows how the BFM, driver, and sequences interact with each other.

### 26.5.1 `sequence.start_sequence()` flow

[Figure 16](#) describes the flow for starting a sequence using the `start_sequence()` method. This flow does not depend on the `driver.bfm_interaction_mode`.



**Figure 16—`sequence.start_sequence()` flow**

For more information on the `start_sequence()` method, see [Table 49](#).

### 26.5.2 do subsequence flow

[Figure 17](#) describes the flow for **do**-ing a subsequence. This flow does not depend on the `driver.bfm_interaction_mode`.

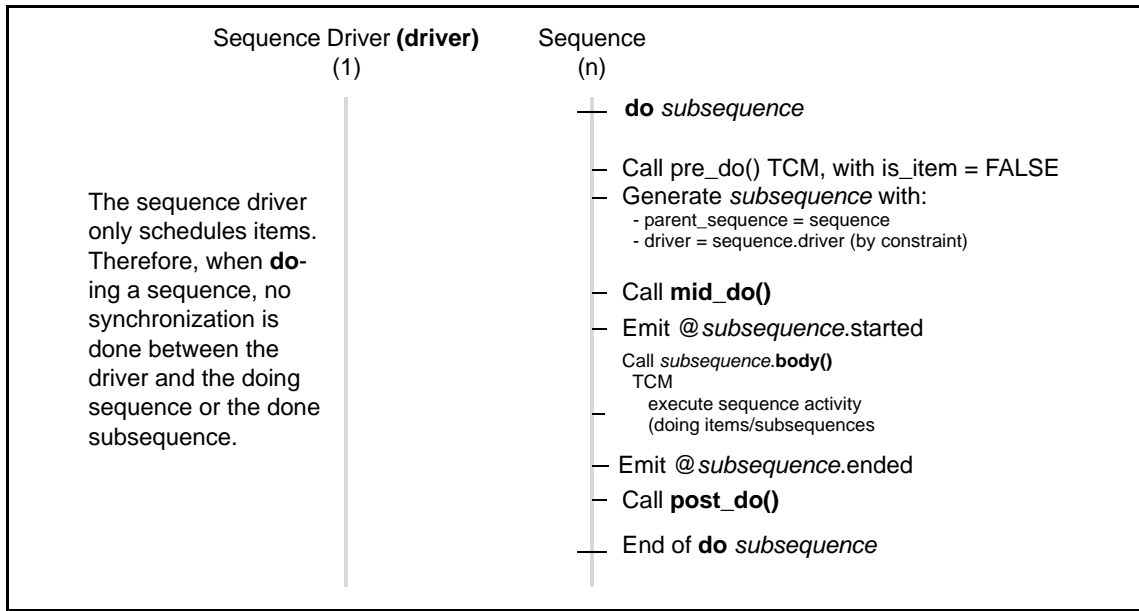


Figure 17—do subsequence flow

For more information on the **do** action, see [26.3](#).

### 26.5.3 do item flow in push mode

[Figure 18](#) describes the flow for **do**-ing an item when *driver.bfm\_interaction\_mode* (see [Table 49](#)) is set to **PUSH\_MODE**.

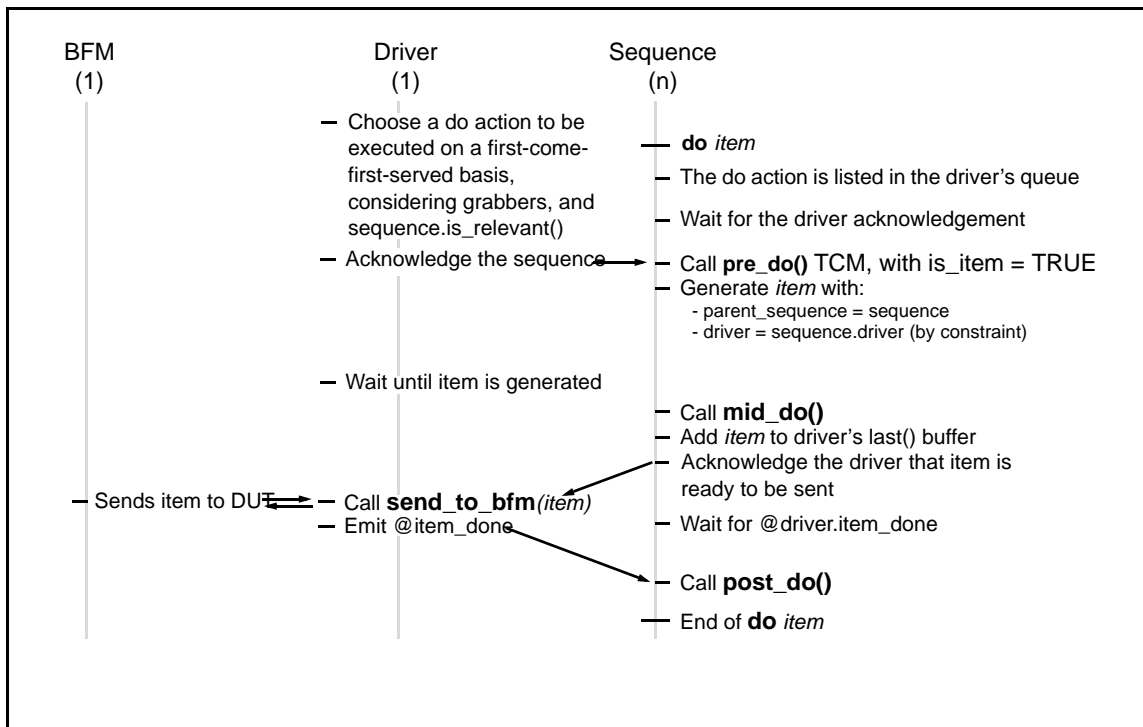


Figure 18—do item flow in push mode

For more information on the **do** action, see [26.3](#).

#### 26.5.4 do item flow in pull mode using `get_next_item()`

[Figure 19](#) describes the flow for **do**-ing an item when `driver.bfm_interaction_mode` (see [Table 49](#)) is set to `PULL_MODE` and `driver.get_next_item()` (see [Table 50](#)) is used to receive items from the driver.

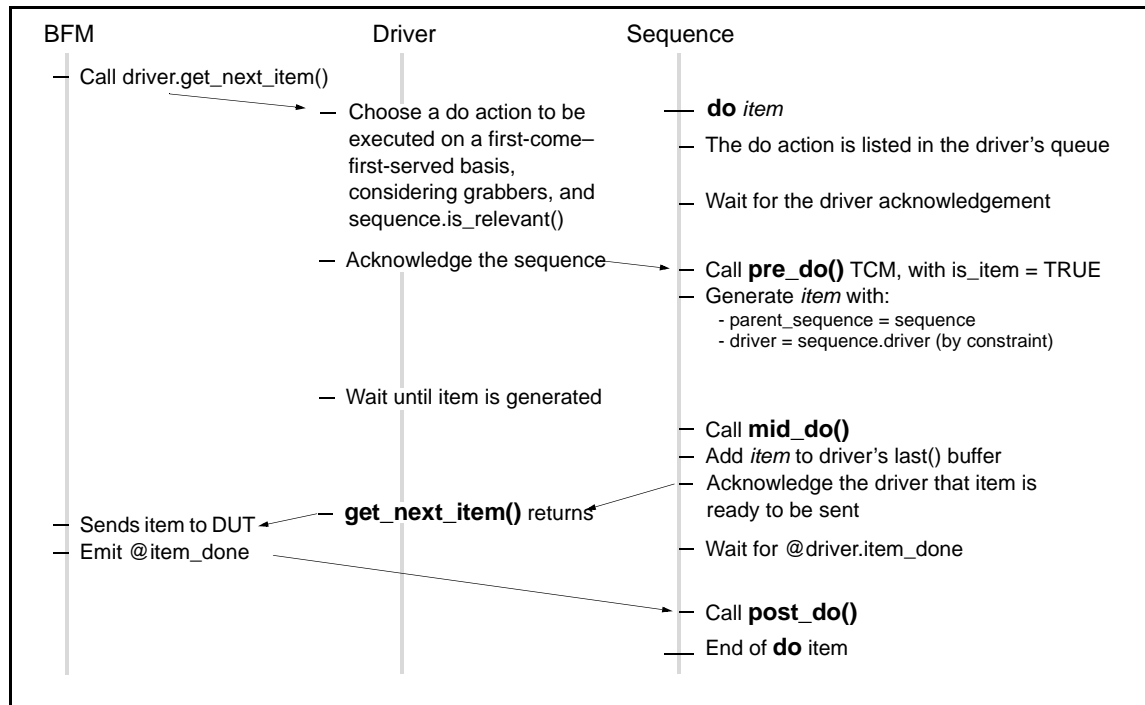
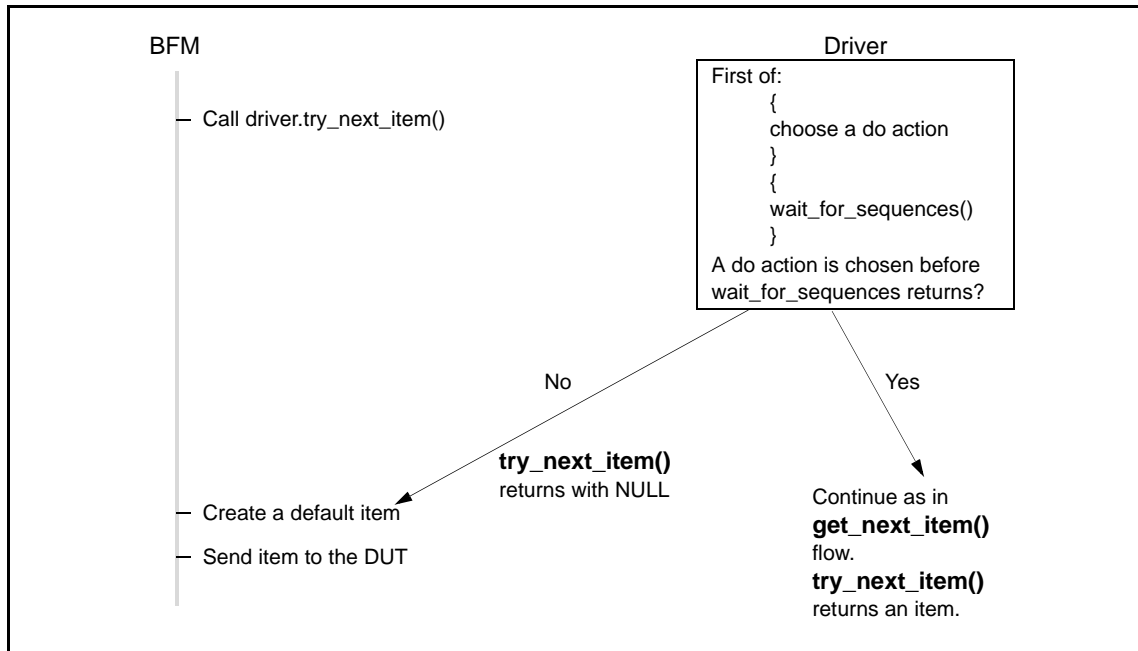


Figure 19—do item flow in pull mode using `get_next_item()`

For more information on the **do** action, see [26.3](#).

#### 26.5.5 do item flow in pull mode using `try_next_item()`

[Figure 20](#) describes the flow for **do**-ing an item when `driver.bfm_interaction_mode` (see [Table 49](#)) is set to `PULL_MODE` and `driver.try_next_item()` (see [Table 50](#)) is used to receive items from the driver. When a **do** is chosen and `pre_do()` takes more than a cycle, `driver.try_next_item()` might also take more than a cycle.



**Figure 20—do item flow in pull mode using try\_next\_item()**

For more information on the **do** action, see [26.3](#).



## 27. List pseudo-methods library

This clause describes the pseudo-methods used to work with lists.

### 27.1 Pseudo-methods overview

A *pseudo-method* is a type of method unique to the *e* language. Pseudo-methods are *e* macros that look like methods. They have the following characteristics:

- Unlike methods, pseudo-methods are not restricted to structs.
- They can be applied to any expression, including literal values, scalars, and compound arithmetic expressions.
- Pseudo-methods cannot be extended.
- Pseudo-methods are defined by using **define as** (see [16.2](#)).
- List pseudo-methods are associated with list data types, as opposed to being within the scope of a struct.

If a method is added that uses the same name as one of the pseudo-methods for a built-in struct, that user-defined method shall take precedence over the built-in struct.

See also [5.1](#), [4.15](#), [4.10.5](#), and [Clause 28](#).

### 27.2 Using list pseudo-methods

A list pseudo-method can be used to operate on a (previously declared) list field or variable by attaching the pseudo-method name, preceded by a period (`.`), to the list name. Any parameters required by the pseudo-method go in parentheses (`()`) after the pseudo-method name.

#### *Example*

The following calls the **apply()** pseudo-method for the list named `p_list`, with the expression `.length + 2` as a parameter. The pseudo-method returns a list of numbers found by adding 2 to the `length` field value in each item in the list.

```
n_list = p_list.apply(.length + 2)
```

Many list pseudo-methods take expressions as parameters and operate on every item in the list. In those pseudo-methods, the **it** variable can be used in an expression to refer to the current list item, and the **index** variable can be used to refer to the current item's list index number.

Pseudo-methods that return values can only be used in expressions.

### 27.3 Pseudo-methods to modify lists

This subclause describes the pseudo-methods that change one or more items in a list.

See also [4.16.2](#), [10.5.1](#), [20.1.2](#), [28.4.1](#), and [29.1.1](#).

### 27.3.1 add(item)

<b>Purpose</b>	Add an item to the end of a list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list.add(item: list-item-type)</i>	
<b>Parameters</b>	<i>list</i>	A list.
	<i>item</i>	An item of list-item type, which is to be added to the list. The item is added at index <code>list.size()</code> , e.g., if the list contains five items, the last item is at index <code>list.size()-1</code> or 4. Adding an item to this list places it at index 5.
<b>Return value</b>	None	

This adds the *item* to the end of the *list*. If the item is a struct, no new struct instance is generated; a pointer to the existing instance of the struct is simply added to the list.

Syntax example:

```
var i_list : list of int;
i_list.add(5)
```

### 27.3.2 add(list)

<b>Purpose</b>	Add a list to the end of another list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list_1.add(list_2: list)</i>	
<b>Parameters</b>	<i>list_1</i>	A list.
	<i>list_2</i>	An item of the same type as <i>list_1</i> , which is to be added to the end of <i>list_1</i> . The list is added at index <code>list.size()</code> , e.g., if the list contains five items, the last item is at index <code>list.size()-1</code> or 4. Adding an item to this list places it at index 5.
<b>Return value</b>	None	

This adds a copy of *list\_2* to the end of *list\_1*.

Syntax example:

```
i_list.add(l_list)
```

### 27.3.3 add0(item)

<b>Purpose</b>	Add an item to the head of a list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list.add0(item: list-type)</i>	
<b>Parameters</b>	<i>list</i>	A list.
	<i>item</i>	An item of the same type as the list items, which is to be added to the head of the list.
<b>Return value</b>	None	

This adds a new item to an existing list. The item is placed at the head of the existing list, as the first position (that is, at index 0). All subsequent items are then reindexed by incrementing their old index by 1.

If the item is a struct, no new struct instance is generated: a pointer to the existing instance of the struct is simply added to the list.

Syntax example:

```
var l_list : list of int = {4; 6; 8};
l_list.add0(2)
```

### 27.3.4 add0(list)

<b>Purpose</b>	Add a list to the head of another list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list_1.add0(list_2: list)</i>	
<b>Parameters</b>	<i>list_1</i>	A list.
	<i>list_2</i>	An item of the same type as <i>list_1</i> , which is to be added to the end of <i>list_1</i> (at <i>list_1</i> index 0).
<b>Return value</b>	None	

This adds a new list to an existing list. A copy of the *list\_2* list is placed at the head of the existing *list\_1* list, starting at the first *list\_1* index. All subsequent items are then reindexed by incrementing their old index by the size of the new list being added.

Syntax example:

```
var i_list : list of int = {1; 3; 5};
var l_list : list of int = {2; 4; 6};
i_list.add0(l_list)
```

### 27.3.5 clear()

<b>Purpose</b>	Delete all items from a list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> .clear()	
<b>Parameters</b>	<i>list</i>	A list.
<b>Return value</b>	None	

This deletes all items in the list.

### 27.3.6 delete()

<b>Purpose</b>	Delete an item from a list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> .delete( <i>index</i> : int)	
<b>Parameters</b>	<i>list</i>	A list.
	<i>index</i>	The index of the item to delete from the list.
<b>Return value</b>	None	

This removes item number *index* from *list* (indexes start counting from 0). The indexes of the remaining items are adjusted to keep the numbering sequential. If the index does not exist in the list, an error shall be issued.

NOTE—*list.delete()* cannot be used to delete a range of items (in a single call).

Syntax example:

```
var i_list : list of int = {2; 4; 6; 8};
i_list.delete(2)
```

### 27.3.7 fast\_delete()

<b>Purpose</b>	Delete an item without adjusting all indexes
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list</i> . <b>fast_delete</b> ( <i>index</i> : int)
<b>Parameters</b>	<i>list</i> A list.
	<i>index</i> The index of the item to delete from the list.
<b>Return value</b>	None

This removes item number *index* from *list* (indexes start counting from 0). The index of the last item in the list is changed to the index of the item that was deleted, so all items following the deleted item keep their original indexes, except the original last index is removed. If the index does not exist in the list, an error shall be issued.

Syntax example:

```
var l_list : list of int = {2; 4; 6; 8};
l_list.fast_delete(2)
```

### 27.3.8 insert(index, item)

<b>Purpose</b>	Insert an item in a list at a specified index
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list</i> . <b>insert</b> ( <i>index</i> : int, <i>item</i> : list-type)
<b>Parameters</b>	<i>list</i> A list.
	<i>index</i> The index in the <i>list</i> where the <i>item</i> is to be inserted.
	<i>item</i> An <i>item</i> of the same type as the <i>list</i> .
<b>Return value</b>	None

This inserts the *item* at the *index* location in the *list*. If *index* is the size of the list, then the *item* is simply added at the end of the list. All indexes in the list are adjusted to keep the numbering correct. If the number of items in the list is smaller than *index*, an error shall be issued.

If the item is a struct, no new struct instance is generated: a pointer to the existing instance of the struct is simply added to the list.

Syntax example:

```
var l_list := {10; 20; 30; 40; 50};
l_list.insert(3, 99)
```

### 27.3.9 insert(index, list)

<b>Purpose</b>	Insert a list in another list starting at a specified index	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list_1.insert(index: int, list_2: list)</i>	
<b>Parameters</b>	<i>list_1</i>	A list.
	<i>index</i>	The index of the position in <i>list_1</i> where <i>list_2</i> is to be inserted.
	<i>list_2</i>	The list to insert into <i>list_1</i> .
<b>Return value</b>	None	

This inserts all items of *list\_2* into *list\_1* starting at *index*. The *index* shall be a positive integer. The size of the new list size is equal to the sum of the sizes of *list\_1* and *list\_2*. If the number of items in *list\_1* is smaller than *index*, an error shall be issued.

Syntax example:

```
var l_list := {10; 20; 30; 40; 50};
var m_list := {11; 12; 13};
l_list.insert(1, m_list)
```

### 27.3.10 pop()

<b>Purpose</b>	Remove and return the last list item	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list.pop()</i> : list-type	
<b>Parameters</b>	<i>list</i>	A list.
<b>Return value</b>	The last item	

This removes the last item (the item at index `list.size() - 1`) in the *list* and returns it. If the list is empty, an error shall be issued.

NOTE—*list.top()* can be used to return the last item in *list* without removing it from the list (see [27.4.24](#)).

Syntax example:

```
var i_list := {10; 20; 30};
var i_item : int;
i_item = i_list.pop()
```

### 27.3.11 pop0()

<b>Purpose</b>	Remove and return the first list item
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list</i> . <b>pop0()</b> : list-type
<b>Parameters</b>	<i>list</i> A list.
<b>Return value</b>	The first item

If the *list* is empty, this method shall issue an error. Otherwise, it removes the first item (the item at index 0) from the *list* and returns that item. It then subtracts 1 from the index of each item remaining in the list.

NOTE—*list.top0()* can be used to return the first item in *list* without removing it from the list (see [27.4.25](#)).

Syntax example:

```
var i_list := {10; 20; 30};
var i_item : int;
i_item = i_list.pop0()
```

### 27.3.12 push()

<b>Purpose</b>	Add an item to the end of a list [same as add(item)]
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list</i> . <b>push(item: list-type)</b>
<b>Parameters</b>	<i>list</i> A list.
	<i>item</i> An item of the same type as the <i>list</i> type, which is to be added to the list. The item is added at index <i>list.size()</i> , e.g., if the list contains five items, the last item is at index <i>list.size()</i> - 1 or 4. Adding an item to this list places it at index 5.
<b>Return value</b>	None

This pseudo-method performs the same function as **add(item)** (see [27.3.1](#)). If the item is a struct, no new struct instance is generated; a pointer to the existing instance of the struct is simply added to the list.

Syntax example:

```
var i_list : list of int;
i_list.push(5)
```

### 27.3.13 push0()

<b>Purpose</b>	Add an item to the head of a list [same as add0(item)]	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>push0</b> ( <i>item</i> : list-type)	
<b>Parameters</b>	<i>list</i>	A list.
	<i>item</i>	An item of the same type as the list items, which is to be added to the head of the list.
<b>Return value</b>	None	

This pseudo-method performs the same function as **add0(item)** (see [27.3.3](#)). If the item is a struct, no new struct instance is generated; a pointer to the existing instance of the struct is simply added to the list.

Syntax example:

```
var l_list : list of int = {4; 6; 8};
l_list.push0(2)
```

### 27.3.14 push(list)

<b>Purpose</b>	Add a list to the end of another list [same as add(item)]	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list_1</i> . <b>push</b> ( <i>list_2</i> : list)	
<b>Parameters</b>	<i>list_1</i>	A list.
	<i>list_2</i>	An item of the same type as <i>list_1</i> , which is to be added to the end of <i>list_1</i> . The list is added at index <code>list.size()</code> , e.g., if the list contains five items, the last item is at index <code>list.size()-1</code> or 4. Adding an item to this list places it at index 5.
<b>Return value</b>	None	

This pseudo-method performs the same function as **add(list)** (see [27.3.2](#)); it adds *list\_2* to the end of *list\_1*.

Syntax example:

```
i_list.push(l_list)
```



### 27.3.15 push0(list)

<b>Purpose</b>	Add a list to the head of another list [same as add0(list)]
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list_1</i> . <b>push0</b> ( <i>list_2</i> : list)
<b>Parameters</b>	<i>list_1</i> A list.
	<i>list_2</i> An item of the same type as <i>list_1</i> , which is to be added to the end of <i>list_1</i> (at <i>list_1</i> index 0).
<b>Return value</b>	None

This pseudo-method performs the same function as **add0(list)** (see [27.3.4](#)); it adds a new list to an existing list. The *list\_2* list is placed at the head of the existing *list\_1* list, starting at the first *list\_1* index. All subsequent items are then reindexed by incrementing their old index by the size of the new list being added.

Syntax example:

```
var i_list : list of int = {1; 3; 5};
var l_list : list of int = {2; 4; 6};
i_list.push0(l_list)
```

### 27.3.16 resize()

<b>Purpose</b>	Change the size of a list
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list</i> . <b>resize</b> ( <i>size</i> : int [, <i>full</i> : bool, <i>filler</i> : exp, <i>keep_old</i> : bool])
<b>Parameters</b>	<i>list</i> A list.
	<i>size</i> A positive integer specifying the desired size.
	<i>full</i> A Boolean value specifying all items are to be filled with filler (defaults to TRUE).
	<i>filler</i> An item of the same type of the list items; used as a filler when <i>full</i> is TRUE.
	<i>keep_old</i> A Boolean value specifying whether to keep existing items already in the list (defaults to FALSE).
<b>Return value</b>	None

This clears the list and increases or decreases the list size according to the new size.

- If only the second parameter, *size*, is used, this method allocates a new list of the given size and all items are initialized to the default value for the list type.
- If any of the three parameters after *size* are used, all three of them shall be used.
- If *full* is TRUE, this method sets all new items to have *filler* as their value.

To resize a list and keep its old values, set both *full* and *keep\_old* to TRUE. If the list is made longer, additional items with the value of *filler* are appended to the list. The following details the behavior of this method for all combinations of *full* and *keep\_old*:

- a) *full* is FALSE, *keep\_old* is FALSE  
An empty list (that is, a list of zero size) is created and memory is allocated for a list of the given *size*.
- b) *full* is TRUE, *keep\_old* is FALSE  
The list is resized to *size* and filled completely with *filler*.
- c) *full* is FALSE, *keep\_old* is TRUE
  - 1) If *size* is greater than the size of the existing list, the list is enlarged to the new *size*, and the new positions are filled with the default value of the list type.
  - 2) If *size* is less than or equal to the size of the existing list, the list is shortened to the new *size*, and all of the existing values up to that size are retained.
- d) *full* is TRUE, *keep\_old* is TRUE
  - 1) If *size* is greater than the size of the existing list, the list is enlarged to the new *size* and the new positions are filled with *filler*.
  - 2) If *size* is less than or equal to the size of the existing list, the list is shortened to the new *size* and all of the existing values up to that size are retained.

Syntax example:

```
var r_list := {2; 3; 5; 6; 8; 9};
r_list.resize(10, TRUE, 1, TRUE)
```

## 27.4 General list pseudo-methods

This subclause describes the syntax for pseudo-methods that perform various operations on lists.

### 27.4.1 apply()

<b>Purpose</b>	Perform a computation on each item in a list
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list</i> . <b>apply</b> ( <i>expr</i> : <i>exp</i> ): <i>list</i>
<b>Parameters</b>	<i>list</i> A list.
	<i>expr</i> Any expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The changed list

This applies the *expr* to each item in the *list* and returns the changed list. The expression *list*.**apply**(**it**,*field*) is the same as *list*.*field* when *field* is a scalar type. The two expressions are different, however, if the field is not a scalar.

*Example*

Assuming `data` is a list of byte, the first expression returns a list containing the first byte of data of each packet item. The second expression is a single item, which is the first item in the concatenated list of all data fields in all packet items.

```
packets.apply(it.data[0]);
packets.data[0]
```

Syntax example:

```
var p_list := {1; 3; 5};
var n_list : list of int;
n_list = p_list.apply(it * 2)
```

**27.4.2 copy()**

<b>Purpose</b>	Make a shallow copy of a list
<b>Category</b>	Predefined method of any struct or unit
<b>Syntax</b>	<i>list</i> . <b>copy()</b> : list
<b>Parameters</b>	<i>list</i> A list.
<b>Return value</b>	None

This is a specific case of *exp*.**copy()** (see [28.4.1](#)), where *exp* is the name of a *list*.

Syntax example:

```
var strlist_1 : list of string = {"A"; "B"; "C"};
var strlist_2 : list of string;
strlist_2 = strlist_1.copy()
```

**27.4.3 count()**

<b>Purpose</b>	Return the number of items that satisfy a given condition
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list</i> . <b>count</b> ( <i>exp</i> : bool): int
<b>Parameters</b>	<i>list</i> A list.
	<i>exp</i> A Boolean expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The number of items

This returns the number of items for which the *exp* is **TRUE**.

Syntax example:

```
var ct : int;
ct = instr_list.count(it.op1 > 200)
```

#### 27.4.4 exists()

<b>Purpose</b>	Check if an index exists in a list
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list</i> . <b>exists</b> ( <i>index</i> : int): bool
<b>Parameters</b>	<i>list</i> A list.
	<i>index</i> An integer expression representing an index to the list.
<b>Return value</b>	A Boolean value

This returns TRUE if an item with the *index* number exists in the *list* or returns FALSE if the index does not exist.

Syntax example:

```
var i_chk : bool;
i_chk = packets.exists(5)
```

#### 27.4.5 first()

<b>Purpose</b>	Get the first item that satisfies a given condition
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list</i> . <b>first</b> ( <i>exp</i> : bool): list-type
<b>Parameters</b>	<i>list</i> A list.
	<i>exp</i> A Boolean expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The first matching item

This returns the first item for which *exp* is TRUE and stops executing.

If there is no such item, the default for the item's type is returned (see [5.1](#)). For a list of scalars, a value of zero (0) is returned if there is no such item. Since zero (0) might be confused with a value found, it is safer to use *list.first\_index()* for lists of scalars.

Syntax example:

```
var i_item : instr;
i_item = instr_list.first(it.op1 > 15)
```

### 27.4.6 first\_index()

<b>Purpose</b>	Get the index of the first item that satisfies a given condition	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>first_index</b> ( <i>exp</i> : bool): int	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	A Boolean expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The index of the first matching item	

This returns the index of the first item for which *exp* is TRUE and stops executing. Otherwise, it returns UNDEF (if there is no such item).

Syntax example:

```
var i_item : int;
i_item = instr_list.first_index(it.op1 > 15)
```

### 27.4.7 get\_indices()

<b>Purpose</b>	Return a sublist of the targeted list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>get_indices</b> ( <i>index-list</i> : list of int): list-type	
<b>Parameters</b>	<i>list</i>	A list.
	<i>index-list</i>	A list of indexes within the list. Each index needs to exist in the list.
<b>Return value</b>	A new list	

This copies the items in *list* that have the indexes specified in *index-list* and returns a new list containing those items. If the *index-list* is empty, an empty list is returned.

Syntax example:

```
var i_list : list of packet;
i_list = packets.get_indices({0; 1; 2})
```

### 27.4.8 has()

<b>Purpose</b>	Check that a list has at least one item that satisfies a given condition	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>has</b> ( <i>exp</i> : bool): bool	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	A Boolean expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	A Boolean value	

This returns **TRUE** if the *list* contains at least one item for which the *exp* is **TRUE**. Otherwise, it returns **FALSE** (if the *exp* is not **TRUE** for any item).

Syntax example:

```
var i_ck : bool;
i_ck = sys.instr_list.has(it.op1 > 31)
```

### 27.4.9 is\_a\_permutation()

<b>Purpose</b>	Check that two lists contain exactly the same items	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list_1</i> . <b>is_a_permutation</b> ( <i>list_2</i> : list): bool	
<b>Parameters</b>	<i>list_1</i>	A list.
	<i>list_2</i>	An list to compare to <i>list_1</i> . This shall be a convertible type of <i>list_1</i> .
<b>Return value</b>	A Boolean value	

This returns **TRUE** if *list\_2* contains the same items as *list\_1*; otherwise, it returns **FALSE** (if any items in one list are not in the other list).

- The order of the items in the two lists does not need to be the same, but the number of items shall be the same for both lists, i.e., items that are repeated in one list shall appear the same number of times in the other list.
- If the lists are lists of structs, *list\_1.is\_a\_permutation(list\_2)* compares the addresses of the struct items, not their contents.
- A *convertible type* is one that automatically converts to match the relevant type.

NOTE—This pseudo-method can be used in a **keep** constraint to fill *list\_1* with the same items contained in the *list\_2*, although not necessarily in the same order.

Syntax example:

```
var lc : bool;
lc = packets_1a.is_a_permutation(packets_1b)
```

### 27.4.10 is\_empty()

<b>Purpose</b>	Check if a list is empty
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list.is_empty()</i> : bool
<b>Parameters</b>	<i>list</i> A list.
<b>Return value</b>	A Boolean value

This returns TRUE if *list* is empty; otherwise, it returns FALSE (if the list is not empty).

Syntax example:

```
var no_l : bool;
no_l = packets.is_empty()
```

### 27.4.11 last()

<b>Purpose</b>	Get the last item that satisfies a given condition
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list.last(exp: bool)</i> : list-type
<b>Parameters</b>	<i>list</i> A list.
	<i>exp</i> A Boolean expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The last matching item

This returns the first item for which *exp* is TRUE and stops executing.

If there is no such item, the default for the item's type is returned (see [5.1](#)). For a list of scalars, a value of zero (0) is returned if there is no such item. Since zero (0) might be confused with a value found, it is safer to use *list.last\_index()* for lists of scalars.

Syntax example:

```
var i_item : instr;
i_item = sys.instr_list.last(it.opl > 15)
```

### 27.4.12 last\_index()

<b>Purpose</b>	Get the index of the last item that satisfies a given condition	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>last_index</b> ( <i>exp</i> : bool): int	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	A Boolean expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The index of the last matching item	

This returns the index of the last item for which *exp* is TRUE and stops executing; otherwise, it returns UNDEF (if there is no such item).

Syntax example:

```
var i_item : int;
i_item = instr_list.last_index(it.op1 > 15)
```

### 27.4.13 max()

<b>Purpose</b>	Get the item with the maximum value of a given expression	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>max</b> ( <i>exp</i> : numeric-type): list-type	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	An integer expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The matching item	

This returns the item for which the *exp* evaluates to the largest value. If more than one item results in the same maximum value, the item latest in the list is returned. If the *list* is empty, an error shall be issued.

Syntax example:

```
var high_item : item_instance;
high_item = item_list.max(it.f_1 + it.f_2)
```



#### 27.4.14 max\_index()

<b>Purpose</b>	Get the index of the item with the maximum value of a given expression	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>max_index</b> ( <i>exp</i> : numeric-type): int	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	An integer expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The index of the matching item	

This returns the index of the item for which the *exp* evaluates to the largest value. If more than one item results in the same maximum value, the index of item latest in the list is returned. If the *list* is empty, an error shall be issued.

Syntax example:

```
var item_index : index;
item_index = sys.item_list.max_index(it.f_1 + it.f_2)
```

#### 27.4.15 max\_value()

<b>Purpose</b>	Return the maximum value found by evaluating a given expression for all items	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>max_value</b> ( <i>exp</i> : numeric-type): exp-type	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	An integer expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The matching value	

This returns the largest integer value found by evaluating the *exp* for every item in the list.

For lists of integer types, [Table 39](#) shows what is returned when the *list* is empty.

**Table 39—Empty list max\_value() return values**

List item type	Value returned
signed integer	MIN_INT (see <a href="#">4.1.4.4</a> )
unsigned integer	zero (0)
long integer	error

Syntax example:

```
var item_val : int;
item_val = sys.item_list.max_value(it.f_1 + it.f_2)
```

#### 27.4.16 min()

<b>Purpose</b>	Get the item with the minimum value of a given expression	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list.min</i> ( <i>exp</i> : numeric-type): list-type	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	An integer expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The matching item	

This returns the item for which the *exp* evaluates to the smallest value. If more than one item results in the same minimum value, the item latest in the list is returned. If the *list* is empty, an error shall be issued.

Syntax example:

```
var low_item : item_instance;
low_item = sys.item_list.min(it.f_1 + it.f_2)
```

#### 27.4.17 min\_index()

<b>Purpose</b>	Get the index of the item with the minimum value of a given expression	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list.min_index</i> ( <i>exp</i> : numeric-type): int	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	An integer expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The index of the matching item	

This returns the index of the item for which the specified *exp* gives the minimal value. If more than one item results in the same minimum value, the index of the item latest in the list is returned. If the *list* is empty, an error shall be issued.

Syntax example:

```
var item_index : index;
item_index = sys.item_list.min_index(it.f_1 + it.f_2)
```

**27.4.18 min\_value()**

<b>Purpose</b>	Return the minimum value found by evaluating a given expression for all items	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>min_value</b> ( <i>exp</i> : numeric-type): exp-type	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	An integer expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The matching value	

This returns the smallest integer value found by evaluating the *exp* for every item in the list.

For lists of integer types, [Table 39](#) shows what is returned when the *list* is empty.

Syntax example:

```
var item_val : int;
item_val = sys.item_list.min_value(it.f_1 + it.f_2)
```

**27.4.19 reverse()**

<b>Purpose</b>	Reverse the order of a list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>reverse</b> (): list	
<b>Parameters</b>	<i>list</i>	A list.
<b>Return value</b>	The changed list	

This returns a new list of all the items in *list* in reverse order.

Syntax example:

```
var s_list := {"A"; "B"; "C"; "D"};
var r_list := s_list.reverse()
```

**27.4.20 size()**

<b>Purpose</b>	Return the size of a list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> .size(): int	
<b>Parameters</b>	<i>list</i>	A list.
<b>Return value</b>	The list size	

This returns an integer equal to the number of items in the *list*. See [10.2.7.3](#) for more information about constraining the size of lists. See also [10.4.1](#) and [4.12.1](#).

NOTE—To control the list size, use a construct like **keep** *list.size() == n*, where *n* is an integer expression. Another way to specify an exact size of a list is by using the *list[n]* index syntax in the list declaration, such as `p_list[n]: list of p`.

Syntax example:

```
print packets.size()
```

**27.4.21 sort()**

<b>Purpose</b>	Sort a list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> .sort( <i>sort-exp</i> : exp): list	
<b>Parameters</b>	<i>list</i>	A list of integers, strings, enumerated items, or Boolean values to sort.
	<i>sort-exp</i>	A scalar or nonscalar expression. The expression can contain references to fields or structs. The <b>it</b> variable can be used to refer to the current list item.
<b>Return value</b>	The changed list	

This returns a new list of all the items in *list*, sorted in increasing order of the values of the *sort-exp*. If the *sort-exp* is a scalar (or string) value, the list is sorted by value. If the *sort-exp* is a nonscalar, the list is sorted by address.

Syntax example:

```
var s_list : list of packet;
s_list = packets.sort(it.f_1 + it.f_2)
```

### 27.4.22 `sort_by_field()`

<b>Purpose</b>	Sort a list of structs by a selected field	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>struct-list</i> . <b>sort_by_field</b> ( <i>field</i> : field-name): list	
<b>Parameters</b>	<i>list</i>	A list of structs.
	<i>field</i>	The name of a field of the list's struct type. Enter the name of the field only, without a preceding period (.) or the term <code>it</code> .
<b>Return value</b>	The changed list	

This returns a new list of all the items in *struct-list*, sorted in increasing order of their *field* values.

NOTE—The *list.sort()* pseudo-method returns the same value as the *list.sort\_by\_field()* pseudo-method, but *list.sort\_by\_field()* is more efficient.

Syntax example:

```
var s_list : list of packet;
s_list = sys.packets.sort_by_field(length)
```

### 27.4.23 `split()`

<b>Purpose</b>	Splits a list at each point where an expression is TRUE	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>split</b> ( <i>split-exp</i> : exp): list of struct-list-holder	
<b>Parameters</b>	<i>list</i>	A list (of any type).
	<i>split-exp</i>	An expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The list of struct-list-holder	

Since *e* does not support lists of lists, this pseudo-method returns a list of type *struct-list-holder*.

- The *struct-list-holder* type is a struct with a single field, `value: list of any-struct`;
- A *struct-list-holder* is a list of structs, with each struct containing a list of items of the original *list* type.
- Each *struct-list-holder* in the returned list contains consecutive items from the *list* that have the same *split-exp* value.

Any fields used in the expression shall be defined in the base type definition, not in **when** subtypes.

Syntax example:

```
var sl_hold := s_list.split(it.f_1 == 16)
```

**27.4.24 top()**

<b>Purpose</b>	Return the last item in a list
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list</i> . <b>top()</b> : list-item
<b>Parameters</b>	<i>list</i> A list.
<b>Return value</b>	The last item

This returns the last item in the *list* without removing it from the list. If the list is empty, an error shall be issued.

Syntax example:

```
var pk : packet;
pk = sys.packets.top()
```

**27.4.25 top0()**

<b>Purpose</b>	Return the first item in a list
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>list</i> . <b>top0()</b> : list-item
<b>Parameters</b>	<i>list</i> A list.
<b>Return value</b>	The first item

This returns the first item in the *list* without removing it from the list. If the list is empty, an error shall be issued.

NOTE—This pseudo-method can be used with **pop0()** to emulate queues.

Syntax example:

```
var pk : packet;
pk = sys.packets.top0()
```

### 27.4.26 unique()

<b>Purpose</b>	Collapse consecutive items that have the same value into one item	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>unique</b> ( <i>select-exp</i> : <i>exp</i> ): list	
<b>Parameters</b>	<i>list</i>	A list of type <i>struct-list-holder</i> .
	<i>split-exp</i>	An expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The changed list	

This returns a new list of all the distinct values in *list*. In the new list, all consecutive occurrences of items for which the value of *exp* are the same are collapsed into one item.

Syntax example:

```
var u_list : list of l_item;
u_list = sys.l_list.unique(it.f_1)
```

### 27.4.27 all()

<b>Purpose</b>	Get all items that satisfy a condition	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>all</b> ( <i>exp</i> : bool): list	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	A Boolean expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	A list of the matching items	

This returns a list of all the items in *list* for which *exp* is TRUE. If no items satisfy the Boolean expression, an empty list is returned. See also [4.16.1](#).

Syntax example:

```
var l_2 : list of packet;
l_2 = sys.packets.all(it.length > 64)
```

### 27.4.28 all\_indices()

<b>Purpose</b>	Get indexes of all items that satisfy a condition	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list.all_indices</i> ( <i>exp</i> : bool): list of int	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	A Boolean expression.
<b>Return value</b>	A list of the indexes for all the matching items	

Returns a list of all indexes of items in *list* for which *exp* is **TRUE**. If no items satisfy the Boolean expression, an empty list is returned.

NOTE—Using **all\_indices()** on an empty list produces another empty list. Trying to use this result in a **gen keeping** constraint can cause a generation contradiction error.

Syntax example:

```
var l_2 : list of int;
l_2 = sys.packets.all_indices(it.length > 5)
```

## 27.5 Math and logic pseudo-methods

This subclause describes the syntax for pseudo-methods that perform arithmetic or logical operations to compute a value using all items in a list.

### 27.5.1 and\_all()

<b>Purpose</b>	Compute the logical AND of all items	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list.and_all</i> ( <i>exp</i> : bool): bool	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	A Boolean expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	A Boolean value	

Returns **TRUE** if all values of the *exp* are true; otherwise, it returns **FALSE** (if the *exp* is false for any item in the *list*). It stops computation once a **FALSE** is established. If the list is empty, this returns **TRUE**.

Syntax example:

```
var bool_val : bool;
bool_val = m_list.and_all(it >= 1)
```



### 27.5.2 or\_all()

<b>Purpose</b>	Compute the logical OR of all items	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>or_all</b> ( <i>exp</i> : bool): bool	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	A Boolean expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	A Boolean value	

This returns a TRUE if any value of the *exp* is true; otherwise, it returns FALSE (if the *exp* is false for every item in the list or the list is empty). It stops computation once a TRUE is established.

Syntax example:

```
var bool_val : bool;
bool_val = m_list.or_all(it >= 100)
```

### 27.5.3 average()

<b>Purpose</b>	Compute the average of an expression for all items	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>average</b> ( <i>exp</i> : numeric-type): numeric-type	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	An integer expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The integer average	

This returns the integer average of the *exp* computed for all the items in the *list*. It returns UNDEF if the list is empty.

Syntax example:

```
var list_ave : int;
list_ave = sys.item_list.average(it.f_1 * it.f_2)
```

#### 27.5.4 product()

<b>Purpose</b>	Compute the product of an expression for all items	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>product</b> ( <i>exp</i> : numeric-type): numeric-type	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	An integer expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The integer product	

This returns the integer product of the *exp* computed over all the items in the *list*. It returns 1 if the list is empty.

Syntax example:

```
var list_prod : int;
list_prod = sys.item_list.product(it.f_1)
```

#### 27.5.5 sum()

<b>Purpose</b>	Compute the sum of all items	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>sum</b> ( <i>exp</i> : numeric-type): numeric-type	
<b>Parameters</b>	<i>list</i>	A list.
	<i>exp</i>	An integer expression. The <b>it</b> variable can be used to refer to the current list item, and the <b>index</b> variable can be used to refer to its index number.
<b>Return value</b>	The integer sum	

This returns the integer sum of the *exp* computed over all the items in the *list*. It returns 0 if the list is empty.

Syntax example:

```
var op_sum : int;
op_sum = sys.instr_list.sum(.opl)
```

### 27.6 List CRC pseudo-methods

This subclause describes the syntax for pseudo-methods that perform CRC (cyclic redundancy check) functions on lists. See also [20.1.1](#) and [20.1.2](#).

### 27.6.1 `crc_8()`

<b>Purpose</b>	Compute the CRC8 of a list of bits or a list of bytes	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>crc_8</b> ( <i>from-byte</i> : int, <i>num-bytes</i> : int): int	
<b>Parameters</b>	<i>list</i>	A list of bits or bytes.
	<i>from-byte</i>	The index number of the starting byte.
	<i>num-bytes</i>	The number of bytes to use.
<b>Return value</b>	The integer value	

This reads the *list* byte-by-byte and returns the integer value of the CRC8 function of a list of bits or bytes. Only the least significant byte is used in the result.

The CRC is computed starting with the *from-byte*, for *num-bytes*. If *from-byte* or *from-byte+num-bytes* is not in the range of the list, an error shall be issued.

NOTE—The algorithm for computing CRC8 is specific for the ATM HEC (Header Error Control) computation. The code used for HEC is a cyclic code with the following generating polynomial:

$$x^{**8} + x^{**2} + x + 1$$

Syntax example:

```
print b_data.crc_8(2, 4)
```

### 27.6.2 `crc_32()`

<b>Purpose</b>	Compute the CRC32 of a list of bits or a list of bytes	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>crc_32</b> ( <i>from-byte</i> : int, <i>num-bytes</i> : int): int	
<b>Parameters</b>	<i>list</i>	A list of bits or bytes.
	<i>from-byte</i>	The index number of the starting byte.
	<i>num-bytes</i>	The number of bytes to use.
<b>Return value</b>	The integer value	

This reads the *list* byte-by-byte and returns the integer value of the CRC32 function of a list of bits or bytes. Only the least significant word is used in the result.

The CRC is computed starting with the *from-byte*, for *num-bytes*. If *from-byte* or *from-byte+num-bytes* is not in the range of the list, an error shall be issued.

NOTE—The algorithm for computing CRC32 generates a 32-bit CRC that is used for messages up to 64 kB in length. Such a CRC can detect 99.99999977% of all errors. The generator polynomial for the 32-bit CRC used for both Ethernet and token ring is:

$$x^{**32} + x^{**26} + x^{**23} + x^{**22} + x^{**16} + x^{**12} + x^{**11} + x^{**10} + x^{**8} + x^{**7} + x^{**5} + x^{**4} + x^{**2} + x + 1$$

Syntax example:

```
print b_data.crc_32(2, 4)
```

### 27.6.3 `crc_32_flip()`

<b>Purpose</b>	Compute the CRC32 of a list of bits or a list of bytes, flipping the bits	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>crc_32_flip</b> ( <i>from-byte</i> : int, <i>num-bytes</i> : int): int	
<b>Parameters</b>	<i>list</i>	A list of bits or bytes.
	<i>from-byte</i>	The index number of the starting byte.
	<i>num-bytes</i>	The number of bytes to use.
<b>Return value</b>	The integer value	

This reads the *list* byte-by-byte and returns the integer value of the CRC32 function of a list of bits or bytes, with the bits flipped. Only the least significant word is used in the result. The bits are flipped as follows:

- The bits inside each byte of the input are flipped.
- The bits in the result are flipped.

The CRC is computed starting with the *from-byte*, for *num-bytes*. If *from-byte* or *from-byte*+*num-bytes* is not in the range of the list, an error shall be issued.

Syntax example:

```
print b_data.crc_32_flip(2, 4)
```

## 27.7 Keyed list pseudo-methods

This subclause describes the syntax for pseudo-methods that can be used only on keyed lists. Using one of these methods on a regular list shall result in an error.

*Keyed lists* are list in which each item has a key associated with it. For a list of structs, the key typically is the name of a particular field in each struct. Each unique value for that field can be used as a key.

- For a list of scalars, the key can be the **it** variable, referring to each item.
- When creating a keyed list, the key shall have a unique value for each item.
- Keyed lists can be searched quickly, by searching on a key value.

### 27.7.1 key()

<b>Purpose</b>	Get the item that has a particular key	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>key</b> ( <i>key-exp</i> : exp): list-item	
<b>Parameters</b>	<i>list</i>	A keyed list.
	<i>key-exp</i>	The key of the item to return.
<b>Return value</b>	The matching list item	

This returns the list item that has the specified key. If there is no such item, the default for the item's type is returned (see 5.1). For a list of scalars, a value of zero (0) is returned if there is no such item. Since zero (0) might be confused with a value found, do not use zero (0) as a key for scalar lists.

Syntax example:

```
var loc_list_item : location;
var i_key          : uint;
i_key = 5;
loc_list_item = locations.key(i_key)
```

### 27.7.2 key\_index()

<b>Purpose</b>	Get the index of an item that has a particular key	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>key_index</b> ( <i>key-exp</i> : exp): int	
<b>Parameters</b>	<i>list</i>	A keyed list.
	<i>key-exp</i>	The key of the item for which the index is to be returned.
<b>Return value</b>	The index of the matching list item	

This returns the integer index of the item that has the specified key; otherwise, it returns UNDEF (if no item with that key exists in the list).

Syntax example:

```
var loc_list_ix : int;
loc_list_ix = locations.key_index(i)
```

### 27.7.3 key\_exists()

<b>Purpose</b>	Check that a particular key is in a list	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<i>list</i> . <b>key_exists</b> ( <i>key-exp</i> : exp): bool	
<b>Parameters</b>	<i>list</i>	A keyed list.
	<i>key-exp</i>	The key for which to search.
<b>Return value</b>	A Boolean value	

This returns `TRUE` if the key exists in the list; otherwise, it returns `FALSE`.

Syntax example:

```
var loc_list_k : bool;
var i          := 5;
loc_list_k = locations.key_exists(i)
```

### 27.7.4 Restrictions on keyed lists

- list.resize()* cannot be used on keyed lists.
- Keyed lists and regular (unkeyed) lists are different types. Assignment is not allowed between a keyed list and a regular list.
- Keyed lists cannot be generated. Trying to generate a keyed list shall result in an error. Therefore, keyed lists need to be defined with the do-not-generate sign (!).

## 28. Predefined methods library

A significant part of *e* functionality is implemented as a set of predefined methods defined directly under the **global** and **sys** structs. Furthermore, every struct inherits a set of predefined methods. Some of these methods can be extended to add functionality and some of them are empty, allowing for user-definition.

Three other predefined structs, **semaphore**, **rdv\_semaphore**, and **locker**, provide predefined methods that are useful in controlling TCMs and in controlling resource sharing between TCMs (see [Clause 32](#)). Then, there are pseudo-methods. Calls to pseudo-methods look like method calls. However, they are associated not with struct expressions, but with other kinds of expressions.

NOTE—Use the **pre\_generate()** or **post\_generate()** methods (see [28.2.2.2](#) and [28.2.2.3](#)) to extend a struct or unit.

### 28.1 Predefined methods of sys

[Table 1](#) (see also [1.4.4](#)) shows the predefined methods of the **sys** struct. These methods are initially empty and intended to be extended by the user. These methods are called by the runtime engine at various phases.

### 28.2 Predefined methods of any\_struct

This subclause defines the methods available for any instantiated user-defined struct or unit.

#### 28.2.1 Setting unit relationships

These methods can be used to get or set unit-related information.

##### 28.2.1.1 get\_unit()

Returns a reference to the unit (see [7.5.1](#)).

##### 28.2.1.2 set\_unit()

Changes the parent unit of a struct (see [7.5.4](#)).

#### 28.2.2 Methods called during execution phases

These methods are used to manipulate structs during execution.

##### 28.2.2.1 The init() method of any\_struct

<b>Purpose</b>	Customize the initialization of a struct
<b>Category</b>	Predefined method of any struct or unit
<b>Syntax</b>	<i>[exp].</i> <b>init()</b>
<b>Parameters</b>	<i>exp</i> An expression that returns a unit or a struct.

The **init()** method of a struct is called when a new instance of the struct is created. This method can be extended to set different values for fields (other than their default values). By default, all fields of scalar type are initialized to zero (0). The initial value of a struct or list is **NULL**; unless the list is a sized list of scalars, in which case, it is initialized to the proper size with each item set to the default value.

The following considerations also apply:

- Initialize the non-generated fields of a struct, especially fields of an enumerated scalar type or unsized lists.
- Enumerated scalar types are initialized to zero (0), even if that is not a legal value for that type.
- Initialize any fields that might be sampled before being assigned.
- Size or initialize any lists that might be filled with data from the DUT.
- Unpacking data from the DUT into an unsized, uninitialized list shall cause a runtime error.
- If a field is initialized, but not marked as non-generated, the initialization is overwritten during generation. To mark a field as non-generated, place a ! character in front of the field name.
- See also [4.12.4](#), [5.1](#), and [20.4.2](#).

Syntax example:

```
init() is also {
    is_ok      = TRUE;
    list_subs  = {320; 330; 340; 350; 360};
    list_color = {black; red; green; blue; yellow; white}
}
```

#### 28.2.2.2 pre\_generate()

Simplifies constraint expressions before they are analyzed by the constraint solver (see [10.5.2](#)).

#### 28.2.2.3 post\_generate()

Derives more complex expressions or values from the generated values (see [10.5.3](#)).

#### 28.2.2.4 The run() method of any\_struct

<b>Purpose</b>	Recommended place for starting TCMs
<b>Category</b>	Method of any struct or unit
<b>Syntax</b>	[ <i>exp</i> .] <b>run()</b>
<b>Parameters</b>	<i>exp</i> An expression that returns a unit or a struct.

When a test is executed, the **global.run\_test()** method calls the **run()** methods of all structs under **sys**, starting from **sys** in depth-first search (DFS) order. After this initial pass, when **any\_struct** is generated (with the **gen** action) or allocated (with **new**), its **run()** method is also invoked. This ensures

- a) The **run()** method of each struct instance is called exactly once, thus avoiding multiple instances of the same started TCM;
- b) TCMs do not start and events do not occur before the *e* program is ready to accept them; and
- c) The **run()** method is called after generation and uses the generated values.

If multiple tests are run in the same session, the **run()** method is called once for each test in the session. The **init()** method is called only once before the first test.

This method can be extended to start user-defined TCMs. The method is initially empty. See also [18.2.2](#).



Syntax example:

```
run() is also {
    start monitor()
}
```

### 28.2.2.5 The quit() method of any\_struct

<b>Purpose</b>	Kill all TCMs, expects, assumes, and events associated with a struct or unit instance
<b>Category</b>	Predefined method of any struct or unit
<b>Syntax</b>	[ <i>exp</i> .] <b>quit()</b>
<b>Parameters</b>	<i>exp</i> An expression that returns a unit or a struct.

This method deactivates a struct instance, killing all TCMs, **expects**, **assumes**, and events associated with the struct and enabling garbage collection. The **quit()** method emits a quit event for that struct instance at the end of the current tick and kills any TCMs, **expects**, **assumes**, and events that were active within the struct in which the **quit()** method is called.

The **quit()** method is called by the **global.stop\_run()** method (see [24.5](#)). It can also be called explicitly.

Syntax example:

```
packet.quit()
```

### 28.2.3 Methods called for customizing specific operations

These methods define how to pack, unpack, or print struct information.

#### 28.2.3.1 do\_pack()

Packs the physical fields of the struct (see [20.4.1.1.1](#)).

#### 28.2.3.2 do\_unpack()

Unpacks a packed list of bit into a struct (see [20.4.1.1.2](#)).

#### 28.2.3.3 The do\_print() method of any\_struct

<b>Purpose</b>	Print struct info
<b>Category</b>	Predefined method of any struct or unit
<b>Syntax</b>	[ <i>exp</i> .] <b>do_print()</b>
<b>Parameters</b>	<i>exp</i> An expression that returns a unit or a struct.

This method controls the printing of information about a particular struct. It can be extended to customize the way information is displayed. This method is called by the **print** action whenever a struct is printed.

Syntax example:

```
do_print() is first {
    outf("Struct %s :", me.s)
}
```

#### 28.2.3.4 The `print_line()` method of `any_struct`

<b>Purpose</b>	Print a struct or a unit in a single line	
<b>Category</b>	Predefined method of any struct or unit	
<b>Syntax</b>	<i>[exp.]</i> <b>print_line</b> ( <b>NULL</b>   <i>struct-type</i> )	
<b>Parameters</b>	<i>exp</i>	An expression that returns a unit or a struct.
	<b>NULL</b> / <i>struct-type</i>	To print a row representation of the struct or unit, the parameter is <b>NULL</b> . To print the header for the list, the parameter is of the form: <i>struct-type</i> .

This method prints lists of structs of a common struct type in a tabulated table format. Each struct in the list is printed in a single line of the table.

There is a limit on the number of fields printed in each line when printing the structs—those fields that fit into a single line are printed—the rest are not printed at all. Each field is printed in a separate column and there is a limitation on the column width. When a field exceeds this width, it is truncated and an asterisk (\*) is placed as the last character of that field's value.

Syntax example:

```
sys.pmi[0].print_line(sys.pmi[0]);
sys.pmi[0].print_line(NULL)
```

### 28.3 Methods and predefined attributes of unit `any_unit`

The predefined methods for **any\_unit** include the following:

- **hdl\_path()**
- **full\_hdl\_path()**
- **e\_path()**
- **agent()**
- **get\_parent\_unit()**

For details about each of these, see [7.4](#).

### 28.4 Pseudo-methods

Pseudo-methods calls look like method calls, but unlike methods they are not associated with structs and are applied to other types of expressions, such as lists. Pseudo-methods cannot be changed or extended through use of the **is only**, **is also**, or **is first** constructs.

### 28.4.1 The `copy()` method of `any_struct`

<b>Purpose</b>	Make a shallow copy
<b>Category</b>	Predefined method of any struct or unit
<b>Syntax</b>	<i>exp</i> . <b>copy()</b> : <i>exp</i>
<b>Parameters</b>	<i>exp</i> Any legal <i>e</i> expression.

This returns a shallow, non-recursive copy of the expression. If the expression is a list or a struct that contains other lists or structs, the second-level items are not duplicated; instead, they are copied by reference. [Table 40](#) details how the copy is made, depending on the type of the expression.

**Table 40—Copying process**

Expression	Procedure
scalar	The scalar value is simply assigned as in a normal assignment.
string	The whole string is copied.
scalar list	A new list with the same size as the original list is allocated and the contents of the original list are duplicated.
list of structs	A new list with the same size as the original list is allocated and the contents of the list is copied by reference, i.e., each item in the new list points to the corresponding item in the original list.
struct	A new struct instance with the same type as the original struct is allocated and all scalar fields are duplicated. All compound fields (lists or structs) in the new struct instance point to the corresponding fields in the original struct.

The following considerations also apply:

- Do not use the assignment operator (=) to copy structs or lists into other data objects. The assignment operator simply manipulates pointers to the data being assigned and does not create new struct instances or lists.
- Use the **deep\_copy()** method (see [29.1.1](#)) to create a recursive copy of a struct or list that contains compound fields or items.

Syntax example:

```
var pmv : packet = sys.pmi.copy()
```

### 28.4.2 `as_a()`

Converts an expression from one data type to another (see [5.8.1](#)).

### 28.4.3 `get_enclosing_unit()`

Returns a reference to the nearest higher-level unit instance of the specified type (see [7.5.2](#)).

#### 28.4.4 to\_string()

<b>Purpose</b>	Convert any expression to a string
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<i>exp.to_string()</i> : string
<b>Parameters</b>	<i>exp</i> A legal <i>e</i> expression.

This method can be used to convert any type to a string; it can be extended for structs and units.

- If the expression is a struct expression, the **to\_string()** method returns a unique identification string for each struct instance, which can be used to reference the struct. By default, the identification string is of the form *type-@ num*, where *num* is a unique struct number over all instances of all structs in the current run.
- If the expression is a list of strings, the **to\_string()** method is called for each element in the list. The string returned contains all the elements, with a newline between each element.
- If the expression is a list of any type except **string**, the **to\_string()** method returns a string containing all the elements, with a space between each element.
- If the expression is a numeric type, it is converted using the current radix with the **radix** prefix.
- If the expression is a string, the **to\_string()** method returns the string.
- If the expression is an enumerated or a Boolean type, the **to\_string()** method returns the value.

Syntax example:

```
print pkts[0].to_string()
```

#### 28.4.5 try\_enclosing\_unit()

Returns a reference to the nearest higher-level unit instance of the specified type, without issuing a runtime error if no unit instance of the specified type is found (see [7.5.3](#)).

### 28.5 Coverage methods

The **covers** struct is a predefined struct containing methods used for coverage and coverage grading. With the exception of the **write\_cover\_file()** method, all of these methods are methods of the **covers** struct:

- **include\_tests()**
- **set\_weight()**
- **set\_at\_least()**
- **set\_cover()**
- **get\_contributing\_runs()**
- **get\_unique\_buckets()**
- **write\_cover\_file()**
- **get\_overall\_grade()**
- **get\_ecov\_name()**
- **get\_test\_name()**
- **get\_seed()**

For details about each of these, see [15.8](#).

## 29. Predefined routines library

Predefined routines are *e* macros that look like methods. The distinguishing characteristics of *predefined routines* are as follows:

- They are not associated with any particular struct.
- They share the same name space for user-defined routines and **global** methods.
- They cannot be modified or extended with the **is only**, **is also**, or **is first** constructs.

See also [18.2](#).

### 29.1 Deep copy and compare routines

The following routines perform recursive copies and comparisons of nested structs and lists. See also [6.11](#).

#### 29.1.1 deep\_copy()

<b>Purpose</b>	Make a recursive copy of a struct and its descendants
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>deep_copy</b> ( <i>struct-inst</i> : exp): struct instance
<b>Parameters</b>	<i>struct-inst</i> An expression that returns a struct instance.

This returns a deep, recursive copy of the struct instance. This routine descends recursively through the fields of a struct and its descendants, copying each field by value, copying it by reference, or ignoring it, depending on the **deep\_copy** attribute set for that field.

The return type of **deep\_copy()** is the same as the declared type of the struct instance.

[Table 41](#) details how the copy is made, depending on the type of the field and the **deep\_copy** attribute (**normal**, **reference**, **ignore**) set for that field. See also [6.11](#).

The following considerations also apply:

- A deep copy of a scalar field (numeric, Boolean, or enumerated) or a string field is the same as a shallow copy performed by a call to **copy()**.
- A struct or list is duplicated no more than once during a single call to **deep\_copy()**.
- If there is more than one reference to a struct or list instance and that instance is duplicated by the call to **deep\_copy()**, every field that referred to the original instance is updated to point to the new instance.
- The **copy()** method of the struct is called by **deep\_copy()**.
- The struct's **copy()** method is called before its descendants are deep copied. If the default **copy()** method is overwritten or extended, this new version of the method is used.
- Add the **reference** attribute to fields that store shared data and to fields that are backpointers (pointers to the parent struct). *Shared data* in this context means data shared between objects inside the deep copy graph and objects outside the deep copy graph. A *deep copy graph* is the imaginary directed graph created by traversing the structs and lists duplicated, where its nodes are the structs or lists and its edges are deep references to other structs or lists.

**Table 41—Copying procedure**

Field type/ attribute	normal	reference	ignore
scalar	The new field holds a copy of the original value.	The new field holds a copy of the original value.	The new field holds a copy of the original value.
scalar list	A new list is allocated with the same size and same elements as the original list.	The new list field holds a copy of the original list pointer. <sup>a</sup>	A new list is allocated with zero size.
struct	A new struct instance with the same type as the original struct is allocated. Each field is copied or ignored, depending on its <b>deep_copy</b> attribute.	The new struct field holds a pointer to the original struct.	No allocation occurs; the field is set to NULL.
list of structs	A new list is allocated with the same number of elements as the original list. New struct instances are also allocated and each field in each struct is copied or ignored, depending on its <b>deep_copy</b> attribute.	The new list field holds a copy of the original list pointer. <sup>a</sup>	A new list is allocated with zero size.

<sup>a</sup>If the list or struct that is pointed to is duplicated (possibly because another field with a normal attribute is also pointing to it), the pointer in this field is updated to point to the new instance. This duplication applies only to instances duplicated by the **deep\_copy()** itself and not to duplications made by the extended/overridden **copy()** method.

Syntax example:

```
var pmv : packet = deep_copy(sys.pmi)
```

### 29.1.2 deep\_compare()

<b>Purpose</b>	Perform a recursive comparison of two struct instances	
<b>Category</b>	Predefined routine	
<b>Syntax</b>	<b>deep_compare</b> ( <i>struct-inst1</i> : exp, <i>struct-inst2</i> : exp, <i>max-diffs</i> : int): list of string	
<b>Parameters</b>	<i>struct-inst1</i> , <i>struct-inst2</i>	An expression returning a struct instance.
	<i>max-diffs</i>	An integer representing the maximum number of differences to report.

This returns a list of strings, where each string describes a single difference between the two struct instances. This routine descends recursively through the fields of a struct and its descendants, comparing each field or ignoring it depending on the **deep\_compare** attribute set for that field.

The two struct instances are “deep equal” if the returned list is empty.

*Deep equal* is defined as follows:

- Two struct instances are deep equal if they are of the same type and all their fields are deep equal.
- Two scalar fields are deep equal if an equality operation applied to them is TRUE.
- Two list instances are deep equal if they are of the same size and all their items are deep equal.

Topology is taken into account. If two non-scalar instances are not in the same location/order in the deep compare graphs, they are not equal. A deep compare graph is the imaginary directed graph created by traversing the structs and lists compared, where its nodes are the structs or lists and its edges are deep references to other structs or lists.

[Table 42](#) details the differences that are reported, depending on the type of the field and the **deep\_compare** attribute (**normal**, **reference**, or **ignore**) set for that field. See also [6.11](#).

**Table 42—Reporting procedure**

Field type/ attribute	normal	reference	ignore
scalar	Their values, if different, are reported.	Their values, if different, are reported.	The fields are not compared.
scalar list	Their sizes, if different, are reported. All items in the smaller list are compared to those in the longer list and their differences are reported.	The fields are equal if their addresses are the same. The items are not compared.	The fields are not compared.
struct	If two structs are not of the same type, their type difference is reported. Also, any differences in common fields is reported. <sup>a, b</sup> If two structs are of the same type, every field difference is reported.	The fields are equal if their addresses are the same. The items are not compared.	The fields are not compared and no differences for them or their descendants are reported.
list of structs	Their sizes, if different, are reported. All structs in the smaller list are deep compared to those in the longer list and their differences are reported.	The fields are equal if their addresses are the same and they point to the same struct instance. <sup>b</sup>	The fields are not compared and no differences for them or their descendants are reported.

<sup>a</sup>Two fields are considered common only if the two structs are the same type, if they are both subtypes of the same base type, or if one is a base type of the other.

<sup>b</sup>If the reference points inside the deep compare graph, a limited topological equivalence check is performed, not just an address comparison.

The difference string reported has the following format:

```
Differences between inst1-id and inst2-id
-----
path:    inst1-value    !=    inst2-value
```

where

<i>path</i>	is a list of field names separated by periods ( . ), from (and not including) the struct instances being compared to the field with the difference.
<i>value</i>	<ul style="list-style-type: none"> <li>a) for scalar field differences, <i>value</i> is the result of <b>out</b>(<i>field</i>).</li> <li>b) for struct field type differences, the type of the field is appended to the path and <i>value</i> is the type of the field.</li> <li>c) for list field size differences, <b>size</b>() is appended to the path and <i>value</i> is the result of <b>out</b>(<i>field.size</i>()).</li> <li>d) for a shallow comparison of struct fields that point outside the deep compare graph, <i>value</i> is the struct address.</li> <li>e) for a comparison of struct fields that point to different locations in the deep compare graphs (topological difference), <i>value</i> is <b>struct#</b> appended to an index representing its location in the deep compare graph.</li> </ul>

NOTE—The same two struct instances or the same two list instances are not compared more than once during a single call to **deep\_compare**().

Syntax example:

```
var diff : list of string = deep_compare(pmi[0], pmi[1], 100)
```

### 29.1.3 deep\_compare\_physical()

<b>Purpose</b>	Perform a recursive comparison of the physical fields of two struct instances
<b>Category</b>	Predefined routine
<b>Syntax</b>	<b>deep_compare_physical</b> ( <i>struct-inst1</i> : exp, <i>struct-inst2</i> : exp, <i>max-diffs</i> : int): list of string
<b>Parameters</b>	<i>struct-inst1</i> , <i>struct-inst2</i> An expression returning a struct instance.
	<i>max-diffs</i> An integer representing the maximum number of differences to report.

Syntax example:

```
var diff : list of string = deep_compare_physical(pmi[0], pmi[1], 100)
```

This returns a list of strings, where each string describes a single difference between the two struct instances. This routine descends recursively through the fields of a struct and its descendants, ignoring all non-physical fields and comparing each physical field or ignoring it, depending on the **deep\_compare\_physical** attribute set for that field.

This routine is the same as the **deep\_compare**() routine (see [29.1.2](#)), except only physical fields (indicated by the % operator prefixed to the field name) are compared.

NOTE—Adding a field under a **when** construct only causes the parent type and the **when** subtype to be different if the added field is a physical field.

## 29.2 Integer arithmetic routines

The following subclauses describe the predefined arithmetic routines in *e*.



**29.2.1 min()**

<b>Purpose</b>	Get the minimum of two numeric values	
<b>Category</b>	Pseudo-routine	
<b>Syntax</b>	<b>min</b> ( <i>x</i> : numeric-type, <i>y</i> : numeric-type): numeric-type	
<b>Parameters</b>	<i>x</i>	A numeric expression.
	<i>y</i>	A numeric expression.

This returns the smaller of the two numeric values.

Syntax example:

```
print min((x + 5), y)
```

**29.2.2 max()**

<b>Purpose</b>	Get the maximum of two numeric values	
<b>Category</b>	Pseudo-routine	
<b>Syntax</b>	<b>max</b> ( <i>x</i> : numeric-type, <i>y</i> : numeric-type): numeric-type	
<b>Parameters</b>	<i>x</i>	A numeric expression.
	<i>y</i>	A numeric expression.

This returns the larger of the two numeric values.

Syntax example:

```
print max((x + 5), y)
```

**29.2.3 abs()**

<b>Purpose</b>	Get the absolute value	
<b>Category</b>	Pseudo-routine	
<b>Syntax</b>	<b>abs</b> ( <i>x</i> : numeric-type): numeric-type	
<b>Parameters</b>	<i>x</i>	A numeric expression.

This returns the absolute value of the expression.

Syntax example:

```
print abs(x)
```

### 29.2.4 odd()

<b>Purpose</b>	Check if an integer is odd
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>odd</b> ( <i>x</i> : numeric-type): bool
<b>Parameters</b>	<i>x</i> A numeric expression.

This returns TRUE if the expression is odd, FALSE if the expression is even.

Syntax example:

```
print odd(x)
```

### 29.2.5 even()

<b>Purpose</b>	Check if an integer is even
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>even</b> ( <i>x</i> : numeric-type): bool
<b>Parameters</b>	<i>x</i> A numeric expression.

This returns TRUE if the expression passed to it is even, FALSE if the expression is odd.

Syntax example:

```
print even(x)
```

### 29.2.6 ilog2()

<b>Purpose</b>	Get the base-2 logarithm
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>ilog2</b> ( <i>x</i> : numeric-type): bool
<b>Parameters</b>	<i>x</i> A numeric expression.

This returns the integer part of the base-2 logarithm of *x*.

Syntax example:

```
print ilog2(x)
```

### 29.2.7 ilog10()

<b>Purpose</b>	Get the base-10 logarithm
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>ilog10</b> ( <i>x</i> : numeric-type): bool
<b>Parameters</b>	<i>x</i> A numeric expression.

This returns the integer part of the base-10 logarithm of *x*.

Syntax example:

```
print ilog10(x)
```

### 29.2.8 ipow()

<b>Purpose</b>	Raise to a power
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>ipow</b> ( <i>x</i> : numeric-type, <i>y</i> : numeric-type): numeric-type
<b>Parameters</b>	<i>x</i> A numeric expression.
	<i>y</i> A numeric expression.

This raises *x* to the power of *y* and returns the result.

Syntax example:

```
print ipow(x, y)
```

### 29.2.9 isqrt()

<b>Purpose</b>	Get the square root
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>isqrt</b> ( <i>x</i> : numeric-type): int
<b>Parameters</b>	<i>x</i> A numeric expression.

This returns the integer part of the square root of *x*.

Syntax example:

```
print isqrt(x)
```

### 29.2.10 div\_round\_up()

<b>Purpose</b>	Division rounded up
<b>Category</b>	Routine
<b>Syntax</b>	<b>div_round_up</b> ( <i>x</i> : int, <i>y</i> : int): int
<b>Parameters</b>	<i>x</i> A numeric expression.
	<i>y</i> A numeric expression.

This returns the result of  $x / y$  rounded up to the next integer. See also [4.9.2](#).

Syntax example:

```
print div_round_up(x, y)
```

## 29.3 Real arithmetic routines

The following arithmetic routines support of **real** type objects:

**Table 43—Arithmetic routines supporting real types**

Routine	Description
floor(real): real	Returns the largest integer that is less than or equal to the parameter.
ceil(real): real	Returns the smallest integer that is greater than or equal to the parameter.
round(real): real	Returns the closest integer to the parameter. In the case of a tie then it returns the integer with the higher absolute value.
log(real): real	Returns the natural logarithm of the parameter.
log10(real): real	Returns the base-10 logarithm of parameter.
pow(real, real): real	Returns the value of the first parameter raised to the power of second one.
sqrt(real): real	Returns the square root of the parameter.
exp(real): real	Returns the value of $e$ raised to the power of the parameter.
sin(real): real	Returns the sine of the parameter given in radians.
cos(real): real	Returns the cosine of the parameter given in radians.
tan(real): real	Returns the tangent of the parameter given in radians.
asin(real): real	Returns the arc sine of the parameter.
acos(real): real	Returns the arc cosine of the parameter.
atan(real): real	Returns the arc tangent of the parameter.

**Table 43—Arithmetic routines supporting real types**

Routine	Description
<code>sinh(real): real</code>	Returns the hyperbolic sine of the parameter.
<code>cosh(real): real</code>	Returns the hyperbolic cosine of the parameter.
<code>tanh(real): real</code>	Returns the hyperbolic tangent of the parameter.
<code>asinh(real): real</code>	Returns the inverse hyperbolic sine of the parameter.
<code>acosh(real): real</code>	Returns the inverse hyperbolic cosine of the parameter.
<code>atanh(real): real</code>	Returns the inverse hyperbolic tangent of the parameter.
<code>atan2(real, real): real</code>	Returns the arc tangent of the two parameters.
<code>hypot(real, real): real</code>	Returns the distance of the point defined by the two parameters from the origin.
<code>is_nan(real): bool</code>	Returns TRUE if the parameter's value is Not-a-Number (NaN).
<code>is_finite(real): bool</code>	Returns TRUE if the parameter's value is a finite real value that is, it is not infinity, negative infinity, or NaN).

NOTE—For integer routines like **ilog()**, **ilog10()**, **ilog2()**, **ipow()**, and **isqrt()**, whose return type is based on the expected type, if the expected type is **real**, then the return type is **int** (bits:\*)).

## 29.4 bitwise\_op()

<b>Purpose</b>	Perform a Verilog-style unary reduction operation
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>bitwise_op</b> ( <i>exp</i> : numeric-type): bit
<b>Parameters</b>	<i>op</i> One of <b>and</b> , <b>or</b> , <b>xor</b> , <b>nand</b> , <b>nor</b> , or <b>xnor</b> .
	<i>exp</i> A numeric expression.

This performs a Verilog-style unary reduction operation on a single operand to produce a single bit result. There is no reduction operator in *e*, but the **bitwise\_op()** routines perform the same functions as reduction operators in Verilog, e.g., **bitwise\_xor()** can be used to calculate parity.

For **bitwise\_nand()**, **bitwise\_nor()**, and **bitwise\_xnor()**, the result is computed by inverting the result of the **bitwise\_and()**, **bitwise\_or()**, and **bitwise\_xor()** operations, respectively. [Table 44](#) shows the predefined pseudo-methods for bitwise operations.

Syntax example:

```
print bitwise_and(b)
```

**Table 44—Bitwise operation pseudo-methods**

Pseudo-method	Operation
<b>bitwise_and()</b>	Boolean AND of all bits
<b>bitwise_or()</b>	Boolean OR of all bits
<b>bitwise_xor()</b>	Boolean XOR of all bits
<b>bitwise_nand()</b>	!bitwise_and()
<b>bitwise_nor()</b>	!bitwise_or()
<b>bitwise_xnor()</b>	!bitwise_xor()

## 29.5 get\_all\_units()

<b>Purpose</b>	Return a list of instances of a specified unit type
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>get_all_units</b> ( <i>unit-type</i> : exp): list of unit type
<b>Parameters</b>	<i>unit-type</i> The name of a unit type, unquoted. The type needs to be defined or an error shall occur.

This routine receives a unit type as a parameter and returns a list of instances of this unit type, as well as any unit instances whose type is contained in the specified unit type.

Syntax example:

```
print get_all_units(XYZ_channel)
```

## 29.6 String routines

None of the string routines in *e* modify the input parameters. When a parameter is passed to one of these routines, the routine makes a copy of the parameter, manipulates the copy, and returns the copy. See also [4.11](#), [5.1.10](#), and [Table 23](#).

### 29.6.1 `append()`

<b>Purpose</b>	Concatenate expressions into a string
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b><code>append()</code></b> : string <b><code>append(item: exp, ...)</code></b> : string
<b>Parameters</b>	<i>item</i> A legal <i>e</i> expression. String expressions shall be enclosed in double quotes (" "). If the expression is a struct instance, the struct ID is printed. If no items are passed to <b><code>append()</code></b> , it returns an empty string.

This calls **`to_string()`** (see [28.4.4](#)) to convert each expression to a string using the current radix setting for any numeric expressions, then it concatenates them and returns the result as a single string.

Syntax example:

```
message = append(list1, " ", list2)
```

### 29.6.2 `appendf()`

<b>Purpose</b>	Concatenate expressions into a string according to a given format
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b><code>appendf(format: string, item: exp, ...)</code></b> : string
<b>Parameters</b>	<i>format</i> A string expression containing a standard C formatting mask for each <i>item</i> (see <a href="#">29.7.3</a> ).
	<i>item</i> A legal <i>e</i> expression. String expressions shall be enclosed in double quotes (" "). If the expression is a struct instance, the struct ID is printed.

This converts each expression to a string. An expression can match either a string format (`%s`) or a numeric format. If it matches a string format, the current radix is used. If it matches a numeric format, that numeric format defines the conversion to a string (see [29.7.3](#)). Once all the expressions are converted, they are concatenated and returned as a single string.

If the number and type of masks in the format string does not match the number and type of expressions, an error shall be issued.

Syntax example:

```
message = appendf("%4d\n %4d\n %4d\n", 255, 54, 1570)
```

### 29.6.3 bin()

<b>Purpose</b>	Concatenate expressions into string, using binary representation for numeric types
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>bin</b> ( <i>item</i> : exp, ...): string
<b>Parameters</b>	<i>item</i> A legal <i>e</i> expression.

This concatenates zero or more expressions into a string, using binary representation for any expressions of numeric types, regardless of the current radix setting. Non-numeric types are converted to a string using `to_string()` (see [28.4.4](#)).

Syntax example:

```
var my_string : string = bin(pi.i, " ", list1, " ", 8)
```

### 29.6.4 dec()

<b>Purpose</b>	Concatenate expressions into string, using decimal representation for numeric types
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>dec</b> ( <i>item</i> : exp, ...): string
<b>Parameters</b>	<i>item</i> A legal <i>e</i> expression.

This concatenates zero or more expressions into a string, using decimal representation for any expressions of numeric types, regardless of the current radix setting. Non-numeric types are converted to a string using `to_string()` (see [28.4.4](#)).

Syntax example:

```
var my_string : string = dec(pi.i, " ", list1, " ", 8)
```

### 29.6.5 hex()

<b>Purpose</b>	Concatenate expressions into string, using hexadecimal representation for numeric types
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>hex</b> ( <i>item</i> : exp, ...): string
<b>Parameters</b>	<i>item</i> A legal <i>e</i> expression.

This concatenates zero or more expressions into a string, using hexadecimal representation for any expressions of numeric types, regardless of the current radix setting. Non-numeric types are converted to a string using `to_string()` (see [28.4.4](#)).



Syntax example:

```
var my_string : string = hex(pi.i, " ", list1, " ", 8)
```

### 29.6.6 quote()

<b>Purpose</b>	Enclose a string in double quotes
<b>Category</b>	Routine
<b>Syntax</b>	<b>quote</b> ( <i>text</i> : string): string
<b>Parameters</b>	<i>text</i> An expression of type <b>string</b> .

This returns a copy of the text, enclosed in double quotes (" "), with any internal quote or backslash preceded by a backslash (\).

Syntax example:

```
out(quote(message))
```

### 29.6.7 str\_chop()

<b>Purpose</b>	Chop the tail of a string
<b>Category</b>	Routine
<b>Syntax</b>	<b>str_chop</b> ( <i>str</i> : string, <i>length</i> : int): string
<b>Parameters</b>	<i>str</i> An expression of type <b>string</b> .
	<i>length</i> An integer representing the desired length.

This removes characters from the end of a string, returning a string of the desired length. If the original string is already less than or equal to the desired length, this routine returns the original string.

Syntax example:

```
var test_dir : string = str_chop(tmp_dir, 13)
```

### 29.6.8 str\_empty()

<b>Purpose</b>	Check if a string is empty
<b>Category</b>	Routine
<b>Syntax</b>	<b>str_empty</b> ( <i>str</i> : string): bool
<b>Parameters</b>	<i>str</i> An expression of type <b>string</b> .

This returns TRUE if the string is empty.

Syntax example:

```
print str_empty(s1)
```

### 29.6.9 str\_exactly()

<b>Purpose</b>	Get a string with exact length
<b>Category</b>	Routine
<b>Syntax</b>	<b>str_exactly</b> ( <i>str</i> : string, <i>length</i> : int): string
<b>Parameters</b>	<i>str</i> An expression of type <b>string</b> .
	<i>length</i> An integer representing the desired length.

This returns a copy of the original string, whose length is the desired length, by adding blanks to the right or by truncating the expression from the right as necessary. If non-blank characters are truncated, the \* character appears as the last character in the string returned.

Syntax example:

```
var long : string = str_exactly("123", 6)
```

### 29.6.10 str\_insensitive()

<b>Purpose</b>	Get a case-insensitive AWK-style regular expression
<b>Category</b>	Routine
<b>Syntax</b>	<b>str_insensitive</b> ( <i>regular_exp</i> : string): string
<b>Parameters</b>	<i>regular_exp</i> An AWK-style regular expression.

This returns an AWK-style regular expression string that is the case-insensitive version of the original regular expression. See also [4.11.2](#).

Syntax example:

```
var insensitive : string = str_insensitive("/hello.*/")
```

### 29.6.11 str\_join()

<b>Purpose</b>	Concatenate a list of strings	
<b>Category</b>	Routine	
<b>Syntax</b>	<b>str_join</b> ( <i>list</i> : list of string, <i>separator</i> : string): string	
<b>Parameters</b>	<i>list</i>	An list of type <b>string</b> .
	<i>separator</i>	The string used to separate the list elements.

This returns a single string that is the concatenation of the strings in the list of strings, separated by the separator. The strings in the list are not changed.

Syntax example:

```
var s := str_join(slist, " - ")
```

### 29.6.12 str\_len()

<b>Purpose</b>	Get string length	
<b>Category</b>	Routine	
<b>Syntax</b>	<b>str_len</b> ( <i>str</i> : string): int	
<b>Parameters</b>	<i>str</i>	An expression of type <b>string</b> .

This returns the number of characters in the original string, not counting the terminating NULL character \0.

Syntax example:

```
var length : int = str_len("hello")
```

### 29.6.13 str\_lower()

<b>Purpose</b>	Convert string to lowercase	
<b>Category</b>	Routine	
<b>Syntax</b>	<b>str_lower</b> ( <i>str</i> : string): string	
<b>Parameters</b>	<i>str</i>	An expression of type <b>string</b> .

This returns a copy of the string with all uppercase characters converted to lowercase.

Syntax example:

```
var lower : string = str_lower("UPPER")
```

### 29.6.14 str\_match()

<b>Purpose</b>	Match strings	
<b>Category</b>	Routine	
<b>Syntax</b>	<b>str_match</b> ( <i>str</i> : string, <i>regular-exp</i> : string): bool	
<b>Parameters</b>	<i>str</i>	An expression of type <b>string</b> .
	<i>regular-exp</i>	An AWK-style or native <i>e</i> regular expression. If not surrounded by slashes (/), the expression is treated as a native style expression (see <a href="#">4.11</a> ).

This returns TRUE if the strings match or FALSE if the strings do not match. The routine **str\_match()** is fully equivalent to the operator `~`. After doing a match, the local pseudo-variables \$1, \$2, ..., \$27 can be used, which correspond to the parenthesized pieces of the match. \$0 stores the entire matched piece of the string. See also [4.10.4](#).

Syntax example:

```
print str_match("ace", "/c(e)?$/")
```

### 29.6.15 str\_pad()

<b>Purpose</b>	Pad string with blanks	
<b>Category</b>	Routine	
<b>Syntax</b>	<b>str_pad</b> ( <i>str</i> : string, <i>length</i> : int): string	
<b>Parameters</b>	<i>str</i>	An expression of type <b>string</b> .
	<i>length</i>	An integer representing the desired length.

This returns a copy of the original string padded with blanks on the right, up to desired length. If the length of the original string is greater than or equal to the desired length, then the original string (not a copy) is returned with no padding.

Syntax example:

```
var s : string = str_pad("hello world", 14)
```

### 29.6.16 str\_replace()

<b>Purpose</b>	Replace a substring in a string with another string	
<b>Category</b>	Routine	
<b>Syntax</b>	<b>str_replace</b> ( <i>str</i> : string, <i>regular-exp</i> : string, <i>replacement</i> : string): string	
<b>Parameters</b>	<i>str</i>	An expression of type <b>string</b> .
	<i>regular-exp</i>	An AWK-style or native <i>e</i> regular expression. If not surrounded by slashes (/), the expression is treated as a native style expression (see <a href="#">4.11</a> ).
	<i>replacement</i>	The string used to replace all occurrences of the regular expression.

A new copy of the original string is created, and then all the matches of the regular expression are replaced by the replacement string. If no match is found, a copy of the source string is returned.

- To incorporate the matched substrings in the *replacement* string, use the backslash escaped numbers: \1, \2, . . .
- In native *e* regular expressions, the portion of the original string that matches the \* or the . . . characters is replaced by the replacement string.
- In AWK-style regular expressions, to replace portions of the regular expressions, mark them with parentheses ( ( ) ).

Syntax example:

```
var s : string = str_replace("crc32", "/(. *32)/", "32_flip")
```

### 29.6.17 str\_split()

<b>Purpose</b>	Split a string to substrings	
<b>Category</b>	Routine	
<b>Syntax</b>	<b>str_split</b> ( <i>str</i> : string, <i>regular-exp</i> : string): list of string	
<b>Parameters</b>	<i>str</i>	An expression of type <b>string</b> .
	<i>regular-exp</i>	An AWK-style or native <i>e</i> regular expression that specifies where to split the string (see <a href="#">4.11</a> ).

This splits the original string on each occurrence of the regular expression and returns a list of strings. If the regular expression occurs at the beginning or the end of the original string, an empty string is returned as the first or last item, respectively. If the regular expression is an empty string, it has the effect of removing all blanks in the original string and the splitting is done on blanks.

The original string is not changed by this operation.

Syntax example:

```
var s : list of string = str_split("first-second-third", "-")
```

### 29.6.18 `str_split_all()`

<b>Purpose</b>	Split a string to substrings, including separators	
<b>Category</b>	Routine	
<b>Syntax</b>	<b><code>str_split_all</code></b> ( <i>str</i> : string, <i>regular-exp</i> : string): list of string	
<b>Parameters</b>	<i>str</i>	An expression of type <b>string</b> .
	<i>regular-exp</i>	An AWK-style or native <i>e</i> regular expression that specifies where to split the string (see <a href="#">4.11</a> ).

This splits the original string on each occurrence of the regular expression and returns a list of strings. If the regular expression occurs at the beginning or the end of the original string, an empty string is returned as the first or last item, respectively. The original string is not changed by this operation.

This routine is similar to **`str_split()`**, except it includes the separators in the resulting list of strings.

Syntax example:

```
var s : list of string = str_split_all(" A B C", "/" + "/")
```

### 29.6.19 `str_sub()`

<b>Purpose</b>	Extract a substring from a string	
<b>Category</b>	Routine	
<b>Syntax</b>	<b><code>str_sub</code></b> ( <i>str</i> : string, <i>from</i> : int, <i>length</i> : int): string	
<b>Parameters</b>	<i>str</i>	An expression of type <b>string</b> .
	<i>from</i>	The index position from which to start extracting. The first character in the string is at index 0.
	<i>length</i>	An integer representing the number of characters to extract.

This returns a copy of a substring of the specified length from the original string, starting from the specified index position. *from* shall be between 0 and *length* + 1 of *str*. If *str* is shorter than *from* + *length*, only the available part is returned.

Syntax example:

```
var dir : string = str_sub("/rtests/test32/tmp", 8, 6)
```

### 29.6.20 str\_upper()

<b>Purpose</b>	Convert a string to uppercase
<b>Category</b>	Routine
<b>Syntax</b>	<b>str_upper</b> ( <i>str</i> : string): string
<b>Parameters</b>	<i>str</i> An expression of type <b>string</b> .

This returns a copy of the original string, converting all lower case characters to upper case characters.

Syntax example:

```
var upper : string = str_upper("lower")
```

## 29.7 Output routines

The predefined output routines print formatted and unformatted information to the screen and to open log files.

### 29.7.1 out()

<b>Purpose</b>	Print expressions to output, with a newline at the end
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>out()</b> <b>out</b> ( <i>item</i> : exp, ...)
<b>Parameters</b>	<i>item</i> A legal <i>e</i> expression. String expressions shall be enclosed in double quotes (" "). If the expression is a struct instance, the struct ID is printed. If no items are passed to <b>out()</b> , an empty string is printed, followed by a newline.

This calls **to\_string()** (see [28.4.4](#)) to convert each expression to a string and prints them to the screen (and to the log file if it is open), followed by a newline.

Syntax example:

```
out("pkts[1].data is ", pkts[1].data)
```

### 29.7.2 `outf()`

<b>Purpose</b>	Print formatted expressions to output, with no newline at the end	
<b>Category</b>	Pseudo-routine	
<b>Syntax</b>	<b><code>outf</code></b> ( <i>format</i> : string, <i>item</i> : exp, ...)	
<b>Parameters</b>	<i>format</i>	A string expression containing a standard C formatting mask for each <i>item</i> (see <a href="#">29.7.3</a> ).
	<i>item</i>	A legal <i>e</i> expression. String expressions shall be enclosed in double quotes (" "). If the expression is a struct instance, the struct ID is printed. If the expression is a list, an error shall be issued.

This converts each expression to a string using the corresponding format string and then prints them to the screen (and to the log file if it is open). For the `%s` mask, `to_string()` (see [28.4.4](#)) is used for creating the string representation of the expression.

- To add a newline, add the `\n` characters to the format string.
- `outf()` can be used to add the newlines where needed.
- Printing of lists is not supported with `outf()`.
- If the number and type of masks in the format string does not match the number and type of expressions, an error shall be issued.

Syntax example:

```
outf("%s %#08x", "pkts[1].data[0] is ", pkts[1].data[0])
```

### 29.7.3 Format string

The format string for the `outf()` and for the `appendf()` routine uses the following syntax:

```
"%[0|-][#][min_width][.max_chars](s|d|x|b|o|u)"
```

where

- `0` pads with 0 instead of blanks. Padding is only done when right alignment is used, on the left end of the expression.
- `-` aligns left. The default is to align right.
- `#` adds 0x before the number. Can be used only with the `x` (hexadecimal) format specifier, e.g., `%#x` or `%#010x`.
- `min_width` is a number that specifies the minimum number of characters. This number determines the minimum width of the field. If there are not enough characters in the expression to fill the field, the expression is padded to make it this many characters wide. If there are more characters in the expression than this number (and if `max_chars` is set large enough), this number is ignored and enough space is used to accommodate the entire expression.
- `max_chars` is a number that specifies the maximum number of characters to use from the expression. Characters in excess of this number are truncated. If this number is larger than `min_width`, then the `min_width` number is ignored. For real number formats `e`, `f`, and `g`, `max_chars` defines the precision—the number of digits after the decimal point.



s	converts the expression to a string. The routine <b>to_string()</b> (see <a href="#">28.4.4</a> ) is used to convert a non-string expression to a string.
d	prints a numeric expression in decimal format.
x	prints a numeric expression in hex format. With the optional # character, adds 0x before the number.
b	prints a numeric expression in binary format.
o	prints a numeric expression in octal format.
u	prints integers ( <b>int</b> and <b>uint</b> ) in uint format.
e	prints a numeric value in the style <code>[ - ]d . dddde?dd</code> where there is one digit before the decimal-point character and the number of digits after it is equal to the precision. If the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears.
f	prints a numeric value in the style <code>[ - ]d . dddde?dd</code> , where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears.
g	prints a numeric value in the in style of either <code>f</code> or <code>e</code> . The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style <code>e</code> is used if the exponent from its conversion is less than <code>-4</code> or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

Printing real values with integer formatting will cause an automatic conversion to `int(bits:*)`.

## 29.8 Operating system interface routines

The routines in this subclause enable use of OS commands from within the *e* programming language. These routines work on all supported OSs.

### 29.8.1 spawn()

<b>Purpose</b>	Send commands to the OS
<b>Category</b>	Pseudo-routine
<b>Syntax</b>	<b>spawn()</b> <b>spawn</b> ( <i>command</i> : string, ...)
<b>Parameters</b>	<i>command</i> An expression of type <b>string</b> .

This takes a variable number of parameters, concatenates them together, and executes the string result as an OS command via **system()** (see [29.8.3](#)).

Syntax example:

```
spawn("touch error.log && ", "grep Error my.elog >error.log")
```

### 29.8.2 spawn\_check()

<b>Purpose</b>	Send a command to the OS and report error
<b>Category</b>	Routine
<b>Syntax</b>	<b>spawn_check</b> ( <i>command</i> : string)
<b>Parameters</b>	<i>command</i> An expression of type <b>string</b> .

This executes a single string as an OS command via **system()** (see [29.8.3](#)), then calls **error()** (see [17.3.2](#)) if the execution of the command returned an error status.

Syntax example:

```
spawn_check("grep Error my.elog >& error.log")
```

### 29.8.3 system()

<b>Purpose</b>	Send a command to the OS
<b>Category</b>	Routine
<b>Syntax</b>	<b>system</b> ( <i>command</i> : string): int
<b>Parameters</b>	<i>command</i> An expression of type <b>string</b> .

This executes the string as an OS command and returns the result. On UNIX systems, the command string is passed to the C `system()` call.

Syntax example:

```
stub = system("cat my.v")
```

### 29.8.4 output\_from()

<b>Purpose</b>	Collect the results of a system call
<b>Category</b>	Routine
<b>Syntax</b>	<b>output_from</b> ( <i>command</i> : string): list of string
<b>Parameters</b>	<i>command</i> An expression of type <b>string</b> .

This executes the string as an OS command and returns the output as a list of string. Under UNIX, **stdout** and **stderr** go to the string list.

Syntax example:

```
log_list = output_from("ls *log")
```

### 29.8.5 output\_from\_check()

<b>Purpose</b>	Collect the results of a system call and check for errors
<b>Category</b>	Routine
<b>Syntax</b>	<b>output_from_check</b> ( <i>command</i> : string): list of string
<b>Parameters</b>	<i>command</i> An expression of type <b>string</b> .

This executes the string as an OS command, returns the output as a list of string, and then calls **error()** (see [17.3.2](#)) if the execution of the command returns an error status. Under UNIX, **stdout** and **stderr** go to the string list.

Syntax example:

```
log_list = output_from_check("ls *.log")
```

### 29.8.6 get\_symbol()

<b>Purpose</b>	Get UNIX environment variable
<b>Category</b>	Routine
<b>Syntax</b>	<b>get_symbol</b> ( <i>env-variable</i> : string): string
<b>Parameters</b>	<i>env-variable</i> An expression of type <b>string</b> .

This returns the environment variable as a string or an empty string if the symbol is not found.

Syntax example:

```
current_display = get_symbol("DISPLAY")
```

### 29.8.7 date\_time()

<b>Purpose</b>	Retrieve current date and time
<b>Category</b>	Routine
<b>Syntax</b>	<b>date_time</b> (): string

This returns the current date and time as a string. The date/time is represented in the standard format supplied by the C library routine `ctime`.

Syntax example:

```
print date_time()
```

### 29.8.8 getpid()

<b>Purpose</b>	Retrieve process ID
<b>Category</b>	Routine
<b>Syntax</b>	<b>getpid()</b> : int

This returns the current process ID as an integer.

Syntax example:

```
print getpid()
```

### 29.9 set\_config()

<b>Purpose</b>	Set values of global configuration parameters	
<b>Category</b>	Predefined routine	
<b>Syntax</b>	<b>set_config</b> ( <i>category</i> : keyword, <i>option</i> : keyword, <i>value</i> : exp [, <i>option</i> : keyword, <i>value</i> : exp...])	
<b>Parameters</b>	<i>category</i>	Is one of the following: <b>cover</b> , <b>gen</b> , <b>memory</b> , <b>print</b> , or <b>run</b> , or any additional implementation-dependent category.
	<i>option</i>	<p>The valid <b>cover</b> options are:</p> <ul style="list-style-type: none"> <li>— <b>mode</b> (either <b>normal</b> or <b>count_only</b>)</li> <li>— <b>absolute_max_buckets</b></li> </ul> <p>The valid <b>generate</b> options are:</p> <ul style="list-style-type: none"> <li>— <b>absolute_max_list_size</b></li> <li>— <b>max_depth</b></li> <li>— <b>max_structs</b></li> </ul> <p>The valid <b>memory</b> options are:</p> <ul style="list-style-type: none"> <li>— <b>gc_threshold</b></li> <li>— <b>gc_increment</b></li> <li>— <b>max_size</b></li> <li>— <b>absolute_max_size</b></li> </ul> <p>The valid <b>print</b> option is: <b>radix</b>.</p> <p>The valid <b>run</b> option is: <b>tick_max</b>.</p> <p>The implementation can also introduce additional options.</p>
	<i>value</i>	The valid values for each option are implementation specific.

This routine sets the configuration options to the specified values.

Syntax example:

```
set_config(memory, gc_threshold, 100M)
```

### 29.10 Random routines

The *e* language supports the following routines to generate random **real** numbers:

**Table 45—Random routines**

Routine	Description
rdist_uniform(from: real, to:real): real	Returns a random <b>real</b> number using uniform distribution in the range <b>from</b> to <b>to</b> .  NOTE—The behavior of rdist_uniform() in <i>e</i> is equivalent to Verilog's \$rdist_uniform() defined in the IEEE1364 (section 17.9.2).



## 30. Predefined file routines library

The global struct named **files** contains predefined routines for working with files. This clause contains information about using files and the predefined file routines. Like most global objects, the predefined routines in the **files** struct cannot be extended with **is first**, **is also**, or **is only**.

### 30.1 File names and search paths

Many of the file routines require a file-name parameter. The following are restrictions on file-name parameters for most routines:

- The file name is taken as a path to the file in the file system. The path can be an absolute pathname or a pathname relative to the current working directory.
- The file name shall not contain any tildes (~), wild card patterns, or environment variables (including the *path env* variable); except for the **files.add\_file\_type()** routine, which accepts tilde (~), wild cards (\*), or the *path env* variable as a *file-name* parameter (see [30.3.1](#)).
- Leave the extension off the file name for files with default extensions, such as .e or .ecom.

NOTE—**files.add\_file\_type()** can be used to verify a valid path to a file exists before using any of the file routines.

### 30.2 File handles

For every open file, a file handle struct exists that contains information about the file. The routine **open()** (see [30.3.4](#)) returns the file handle as a variable of type **file**. The name of the file variable is used in low-level file operations such as the **files.read()**, **files.write()**, and **files.flush()** routines (see [30.3](#)).

### 30.3 Low-level file methods

This subclause contains descriptions of the file methods that use file handle structs.

#### 30.3.1 add\_file\_type()

<b>Purpose</b>	Get a file name	
<b>Category</b>	Method	
<b>Syntax</b>	<b>files.add_file_type</b> ( <i>file-name</i> : string, <i>file-ext</i> : string, <i>exists</i> : bool): string	
<b>Parameters</b>	<i>file-name</i>	The name of the file to access. A wild card pattern can be used.
	<i>file-ext</i>	The file extension, including the dot (.). This can be empty.
	<i>exists</i>	Whether to check for existence of the file.

This assigns a string consisting of *file-name* and *file-ext* to a **string** variable. The *file-name* can contain ~, the *path env* variable, and \* wild cards. The \* wild card represents any combination of ASCII characters.

If *file-name* already contains an extension, then *file-ext* is ignored. If *file-ext* is empty, the *file-name* is used with no extension. If *exists* is FALSE, the method returns the file-name string without checking for the existence of the file. Wild cards, ~, and the *path env* variable are not evaluated in this case.

If *exists* is TRUE, the *e* program checks to see if there is a file that matches the *file-name* in the current directory, based on the following rules:

- a) If there is one and only one file that matches the *file-name* pattern, the file's name is returned.
- b) If there is no match in the current directory, then the *path env* directories are searched for the file.  
If there are multiple matching files in different directories within the *path env* variable, the first one found is returned.
- c) If no matching file can be found or if more than one file is found in a directory that matches a wild card, an error shall be issued.

Syntax example:

```
var fv : string;
fv = files.add_file_type("fname", ".e", FALSE)
```

### 30.3.2 close()

<b>Purpose</b>	Close a file
<b>Category</b>	Method
<b>Syntax</b>	<b>files.close</b> ( <i>file</i> : file-handle)
<b>Parameters</b>	<i>file</i> The file handle of the file to be closed.

This flushes the file buffer and closes the *file*. The file needs to have been previously opened using **open()**.

NOTE—Close a file when no further activity is planned for it to prevent unintentional operations on its contents.

Syntax example:

```
files.close(f_desc)
```

### 30.3.3 flush()

<b>Purpose</b>	Flush file buffers
<b>Category</b>	Method
<b>Syntax</b>	<b>files.flush</b> ( <i>file</i> : file-handle)
<b>Parameters</b>	<i>file</i> The file handle of the file to flush.

File data is buffered in memory and only written to disk at certain times, such as when the file is closed. This method causes data to be written to the disk immediately, which can be useful if two processes are using the same disk file, e.g., to ensure the current data from one process is written to the file before the other process reads from the file.

Syntax example:

```
files.flush(a_file)
```



### 30.3.4 open()

<b>Purpose</b>	Open a file for reading or writing or both	
<b>Category</b>	Method	
<b>Syntax</b>	<b>files.open</b> ( <i>file-name</i> : string, <i>mode</i> : string, <i>file-role</i> : string): file	
<b>Parameters</b>	<i>file-name</i>	The name of the file to open. Wild cards, ~, and the <i>path env</i> variable are not allowed in the file name; to use them to select files, see <a href="#">30.3.1</a> .
	<i>mode</i>	The read/write mode for the file. The mode can be one of the following: <b>r</b> —Open the file for reading. <b>w</b> —Open the file for writing (overwrite the existing contents). <b>rw</b> —Open the file for reading and writing (add to the end of the existing contents). <b>a</b> —Open the file for appending (add to the end of the existing contents).
	<i>file-role</i>	A text description used in error messages about the file.

This opens the file for reading, writing, both reading and writing, or appending, according to the mode (**r**, **w**, **rw**, or **a**, respectively) and returns the file handle of the file. The *file-role* is a description of the file, e.g., "source file."

An error shall be issued only when a file is required for reading, but cannot be found (mode **r**). No error is reported when the file is opened, but not found (all other modes: **rw**, **w**, and **a**).

Syntax example:

```
var m_file : file;
m_file = files.open("a_file.txt", "r", "Text File")
```

### 30.3.5 read()

<b>Purpose</b>	Read an ASCII line from a file	
<b>Category</b>	Method	
<b>Syntax</b>	<b>files.read</b> ( <i>file</i> : file-handle, <i>string-var</i> : *string): bool	
<b>Parameters</b>	<i>file</i>	The file handle of the file that contains the text to read.
	<i>string-var</i>	The variable used to hold the read ASCII text.

This reads a line of text from a file into a string variable. The file shall have been opened with **open()**. The line from the file (without the final `\n` newline character) is read into the variable. This method returns **TRUE** on success. If the method cannot read a line (e.g., if the end of the file is reached), it returns **FALSE**. See also [5.8](#) for information about type conversion between scalar types.

Syntax example:

```
r_b = files.read(f_desc, m_string)
```

### 30.3.6 read\_lob()

<b>Purpose</b>	Read from a binary file into a list of bits	
<b>Category</b>	Method	
<b>Syntax</b>	<b>files.read_lob</b> ( <i>file</i> : file-handle, <i>size-in-bits</i> : int): list of bit	
<b>Parameters</b>	<i>file</i>	The file handle of the file from which to read.
	<i>size-in-bits</i>	The number of bits to read (in multiples of 8).

This reads data from a binary file into a list of bits and returns the list of bits. The file shall already have been opened with **open()**. To read an entire file, use UNDEF as the *size-in-bits*. See also [5.8](#) for information about type conversion between scalar types.

Syntax example:

```
var m_file : file = files.open("a_file.dat", "r", "Data");
var b_l    : list of bit;
b_l = files.read_lob(m_file, 32)
```

### 30.3.7 write()

<b>Purpose</b>	Write a string to file	
<b>Category</b>	Method	
<b>Syntax</b>	<b>files.write</b> ( <i>file</i> : file-handle, <i>text</i> : string)	
<b>Parameters</b>	<i>file</i>	The file handle of the file in which to write ( <b>w</b> or <b>a</b> ).
	<i>text</i>	The text to write to the file.

This adds a string to the end of an existing, open file. A newline `\n` is added automatically at the end of the string.

The file shall already have been opened with **open()**, otherwise an error shall be issued. If the number of items in the formatting mask is different from the number of item expressions, an error shall be issued.

How the data is written to the file is affected by the **open()** mode (w or a) and whether or not the file already exists, as follows:

- If the file did not previously exist and the w (write) option is used with **open()**, then **write()** writes the data into a new file.
- If the file did not previously exist and the a (append) option is used with **open()**, then no data is written.
- If the file did previously exist and the w (write) option is used with **open()**, then **write()** overwrites the contents of the file.
- If the file did previously exist and the a (append) option is used with **open()**, then **write()** appends the data to the existing contents of the file.

NOTE—The Perl-style `>>` append operator can be prefixed to the name of the file to open the file for an append-write.

Syntax example:

```
files.write(m_file, "Test Procedure")
```

### 30.3.8 write\_lob()

<b>Purpose</b>	Write a list of bits to a binary file
<b>Category</b>	Method
<b>Syntax</b>	<b>files.write_lob</b> ( <i>file</i> : file-handle, <i>bit-list</i> : list of bit)
<b>Parameters</b>	<i>file</i> The file handle of the file in which to write.
	<i>bit-list</i> A list of bits to write to the file. The size of the list shall be a multiple of 8.

This writes all the bits in the bit list (whose size shall be a multiple of 8) to the end of the file specified by *file*. The file shall already have been opened with **open()**.

Lists of bits are always written in binary format.

For more details on how files are written, see [30.3.7](#).

Syntax example:

```
var m_file : file = files.open("a_f.dat", "w", "My data");
var b_l    : list of bit;
files.write_lob(m_file, b_l)
```

### 30.3.9 writef()

<b>Purpose</b>	Write to a file in a specified format
<b>Category</b>	Pseudo-method
<b>Syntax</b>	<b>files.writef</b> ( <i>file</i> : file-handle, <i>format</i> : string, <i>item</i> : exp, ...)
<b>Parameters</b>	<i>file</i> The file handle of the file in which to write.
	<i>format</i> A string containing a standard C formatting mask (see <a href="#">29.7.3</a> ) for each <i>item</i> .
	<i>item</i> An <i>e</i> expression to write to the file.

This adds a formatted string to the end of the specified file. No newline is automatically added. (Use `\n` in the formatting mask to add a newline.)

For more details on how files are written, see [30.3.7](#).

See also [29.7.3](#).

Syntax example:

```
var m_file : file = files.open("a_f.dat", "w", "My data");
var b_l    : list of bit;
files.write_lob(m_file, b_l)
```

## 30.4 General file routines

This subclause contains descriptions of the general filing routines. See also [5.8](#) for information about type conversion between scalar types.

### 30.4.1 file\_age()

<b>Purpose</b>	Get a file's modification date
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_age</b> ( <i>file-name</i> : string): int
<b>Parameters</b>	<i>file-name</i> The file whose age is to be found.

This returns the modification date of the file as an integer. This routine can be used to compare the modification dates of files. The integer returned by the routine is not recognizable as a date, but is a unique number derived from the file's modification date. If the modification date includes the time of day, the time is factored into the number the routine returns. Newer files produce larger numbers than older files.

If the file does not exist, an error shall be issued.

Syntax example:

```
var f_data : int;
f_data = files.file_age("f.txt")
```

### 30.4.2 file\_append()

<b>Purpose</b>	Append files
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_append</b> ( <i>from-file-name</i> : string, <i>to-file-name</i> : string)
<b>Parameters</b>	<i>from-file-name</i> The name of the file to append.
	<i>to-file-name</i> The name of the file where <i>from-file-name</i> is appended.

This adds the contents of the file named *from-file-name* to the end of the file named *to-file-name*. If either of the files does not exist, an error shall be issued.

Syntax example:

```
files.file_append(f_1, f_2)
```

### 30.4.3 file\_copy()

<b>Purpose</b>	Create a copy of a file
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_copy</b> ( <i>from-file-name</i> : string, <i>to-file-name</i> : string)
<b>Parameters</b>	<i>from-file-name</i> The name of the file to copy.
	<i>to-file-name</i> The name of the (new) copy file.

This makes a copy of *from-file-name*, using the name *to-file-name*. If a file already exists with the *to-file-name*, the contents of that file are replaced by the contents of the file named *from-file-name*. If the file named *from-file-name* does not exist, an error shall be issued.

Both parameters shall be file names (using an absolute pathname or a pathame relative to the current working directory). If a directory name is given as a parameter, an error shall be issued.

Syntax example:

```
files.file_copy("file_1.txt", "tmp_file.txt")
```

### 30.4.4 file\_delete()

<b>Purpose</b>	Delete a file
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_delete</b> ( <i>file-name</i> : string)
<b>Parameters</b>	<i>file-name</i> The file to delete.

This deletes the specified file. If the file cannot be found, an error shall be issued.

Syntax example:

```
files.file_delete("run_1.log")
```

### 30.4.5 file\_exists()

<b>Purpose</b>	Check if a file exists
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_exists</b> ( <i>file-name</i> : string): bool
<b>Parameters</b>	<i>file-name</i> The file to check.

This checks if the *file-name* exists in the file system. It returns `TRUE` if the file exists or issues an error if it does not exist. It also returns `TRUE` if the file is a directory. The routine does not check whether the file is readable or not.

NOTE—This routine only checks for the existence of the specified file; for a routine that can check for multiple similarly named files, see [30.3.1](#).

Syntax example:

```
var f_e : bool;
f_e = files.file_exists("file_1.e")
```

#### 30.4.6 file\_extension()

<b>Purpose</b>	Get the extension of a file
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_extension</b> ( <i>file-name</i> : string): string
<b>Parameters</b>	<i>file-name</i> The file name.

This returns a string containing the file extension, which is the sequence of characters after the last period (.) in the file name.

Syntax example:

```
var f_ext : string;
f_ext = files.file_extension("f_1.exe")
```

#### 30.4.7 file\_is\_dir()

<b>Purpose</b>	Check if a file is a directory
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_is_dir</b> ( <i>file-name</i> : string): bool
<b>Parameters</b>	<i>file-name</i> The file to check.

This returns `TRUE` if the file exists and is a directory. Otherwise, it returns `FALSE` (if the file does not exist or is not a directory).

Syntax example:

```
var is_d : bool;
is_d = files.file_is_dir("a_fil")
```

### 30.4.8 file\_is\_link()

<b>Purpose</b>	Check if a file is a symbolic link
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_is_link</b> ( <i>file-name</i> : string): bool
<b>Parameters</b>	<i>file-name</i> The file to check.

This returns `TRUE` if the file exists and is a symbolic link. Otherwise, it returns `FALSE` (if the file does not exist or is not a symbolic link).

Syntax example:

```
var is_l : bool;
is_l = files.file_is_link("a_fil")
```

### 30.4.9 file\_is\_readable()

<b>Purpose</b>	Check if a file is readable
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_is_readable</b> ( <i>file-name</i> : string): bool
<b>Parameters</b>	<i>file-name</i> The file to check.

This returns `TRUE` if the file exists and is readable. Otherwise, it returns `FALSE` (if the file does not exist or is not readable).

Syntax example:

```
var is_rd : bool;
is_rd = files.file_is_readable("a_fil")
```

### 30.4.10 file\_is\_regular()

<b>Purpose</b>	Check if a file is a regular file (not a directory or link)
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_is_regular</b> ( <i>file-name</i> : string): bool
<b>Parameters</b>	<i>file-name</i> The file to check.

This returns `TRUE` if the file exists and is a regular file. Otherwise, it returns `FALSE` (if the file does not exist, or it is a directory or symbolic link).

Syntax example:

```
var is_rg : bool;
is_rg = files.file_is_regular("a_fil")
```

### 30.4.11 file\_is\_temp()

<b>Purpose</b>	Check if a file is a temporary file
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_is_temp</b> ( <i>file-name</i> : string): bool
<b>Parameters</b>	<i>file-name</i> The file to check.

This returns TRUE if the file is a temporary file per the convention of the host OS. Otherwise, it returns FALSE.

Syntax example:

```
var is_tmp : bool;
is_tmp = files.file_is_temp("a_fil")
```

### 30.4.12 file\_is\_text()

<b>Purpose</b>	Check if a file is a text file
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_is_text</b> ( <i>file-name</i> : string): bool
<b>Parameters</b>	<i>file-name</i> The file to check.

This returns TRUE if the file is a text file (i.e., it contains more than 20% printable characters). Otherwise, it returns FALSE (if the file does not exist or it is not a text file). The following considerations also apply:

- Characters are deemed printable based on the ANSI C function `isprint()`.
- At least the first 80 bytes of a file shall be checked in determining “printability.”

Syntax example:

```
var is_txt : bool;
is_txt = files.file_is_text("a_fil")
```



### 30.4.13 file\_rename()

<b>Purpose</b>	Rename a file
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_rename</b> ( <i>from-file-name</i> : string, <i>to-file-name</i> : string)
<b>Parameters</b>	<i>from-file-name</i> The file to rename.
	<i>to-file-name</i> The new file name.

This renames the file named *from-file-name* to *to-file-name*. If any files already exists with *to-file-name*, that file is overwritten by the contents of the file named *from-file-name*.

If the file or directory is not writable, an error shall be issued.

Syntax example:

```
files.file_rename("f_1.exe", "b_1.exe")
```

### 30.4.14 file\_size()

<b>Purpose</b>	Get the size of a file
<b>Category</b>	Method
<b>Syntax</b>	<b>files.file_size</b> ( <i>file-name</i> : string): int
<b>Parameters</b>	<i>file-name</i> The file name.

This returns the integer number of bytes in the file. If the file does not exist, an error shall be issued.

Syntax example:

```
var f_s : int;
f_s = files.file_size("a_file.txt")
```

### 30.4.15 new\_temp\_file()

<b>Purpose</b>	Create a unique temporary file name
<b>Category</b>	Method
<b>Syntax</b>	<b>files.new_temp_file</b> (): string

This computes a file name [a string with a period (.) at the end]. Each file name this routine produces contains the name of the process, so names are unique across processes. The files are saved in the temporary files directory (set per the convention of the host OS).

NOTE—This routine only creates the file name; to create a file (with this name), see [30.3.4](#).

Syntax example:

```
var t_name : string;
t_name = files.new_temp_file()
```

### 30.4.16 write\_string\_list()

<b>Purpose</b>	Write a list of strings to a file
<b>Category</b>	Method
<b>Syntax</b>	<b>files.write_string_list</b> ( <i>file-name</i> : string, <i>strings</i> : list of string)
<b>Parameters</b>	<i>file-name</i> The file name in which to write.
	<i>strings</i> A list of strings to write to the file.

This writes a list of strings into a file. Every string is written on a separate line in the file, with `\n` appended to the end of the string. If the file already exists, it is overwritten.

If the list of strings contains a `NULL`, an error shall be issued.

NOTE—The Perl-style `>>` append operator can be prefixed to the name of the file to open the file for an append-write.

Syntax example:

```
var s_list := {"a string"; "another string"};
files.write_string_list("a_file.txt", s_list)
```

## 30.5 Reading and writing structs

This subclause contains descriptions of the file routines that use read structs from files and write structs to files. Structs in *e* can be read from files and written to files in either binary or ASCII format.

### 30.5.1 read\_ascii\_struct()

<b>Purpose</b>	Read ASCII file data into a struct
<b>Category</b>	Method
<b>Syntax</b>	<b>files.read_ascii_struct</b> ( <i>file-name</i> : string, <i>struct</i> : string): struct
<b>Parameters</b>	<i>file-name</i> The name of the (ASCII) file to read.
	<i>struct</i> The string in which to read the data.

This reads the ASCII contents of *file-name* into a struct of type *struct* and returns a struct. The struct being read needs to be cast to the correct data type (see [5.8.1](#)). If the file does not exist, an error shall be issued.

Syntax example:

```
var a_str : s_struct;
a_str = files.read_ascii_struct("a_s.out", "s_struct").as_a(s_struct)
```

### 30.5.2 read\_binary\_struct()

<b>Purpose</b>	Read the contents of a binary file into a struct	
<b>Category</b>	Pseudo-method	
<b>Syntax</b>	<b>files.read_binary_struct</b> ( <i>file-name</i> : string, <i>struct</i> : string, <i>check-version</i> : bool): struct	
<b>Parameters</b>	<i>file</i>	The name of the file to read. The file shall have been created by using <b>write_binary_struct()</b> .
	<i>struct</i>	The string in which to read the data.
	<i>check-version</i>	Set to TRUE to compare the contents of the file being read with the definition of the struct in the currently running module. Set to FALSE to allow minor changes.

This reads the binary contents of *file-name* into a struct of the specified type and returns a struct. The struct being read needs to be cast to the correct data type (see [5.8.1](#)).

If *check-version* is FALSE, the routine can run even if the order of fields in the file struct is different from the order of fields in the currently running *e* module. If *check-version* is TRUE, an error shall be issued if the struct definition has been changed in any way since the struct was written to the file.

Syntax example:

```
var b_str : s_struct;
b_str = files.read_binary_struct("b.out", "s_struct", TRUE).as_a(s_struct)
```

### 30.5.3 write\_ascii\_struct()

<b>Purpose</b>	Write the contents of a struct to a file in ASCII format	
<b>Category</b>	Struct member	
<b>Syntax</b>	<b>files.write_ascii_struct</b> ( <i>file-name</i> : string, <i>struct</i> : struct, <i>comment</i> : string, <i>indent</i> : bool, <i>depth</i> : int, <i>max-list-items</i> : int)	
<b>Parameters</b>	<i>file-name</i>	The name of the file in which to write. The default extension is <i>.erd</i> , which stands for <i>e</i> -readable data.
	<i>struct</i>	The name of the struct instance to write to the file.
	<i>comment</i>	A string for a comment at the beginning of the file.
	<i>indent</i>	A Boolean selector for indentation to the struct's field depth.
	<i>depth</i>	The number of levels of nested structs to write.
	<i>max-list-items</i>	For lists, how many items from each list to write.

This recursively writes the contents of the *struct* to the *file-name* in ASCII format. If the struct contains other structs, those structs are also written to the file. If the number of hierarchical levels contained in the *struct* is greater than the specified *depth*, levels below the *depth* level are represented by ellipses ( . . . ) in the ASCII file.

- The `.erd` default file name extension is automatically added to the file name only if the specified file name has no extension and does not end with a period (`.`), e.g., `myfile` becomes `myfile.erd`.
- This routine does not write any of the *e* program internal structs. It can write the `sys` struct, but not any predefined structs within `sys`.
- If the file already exists, it is overwritten.

Syntax example:

```
files.write_ascii_struct("a_file.dat", a_str, "my_struct", TRUE, 2, 10)
```

### 30.5.4 write\_binary\_struct()

<b>Purpose</b>	Write the contents of a struct to a file in binary format	
<b>Category</b>	Method	
<b>Syntax</b>	<b>files.write_binary_struct</b> ( <i>file-name</i> : string, <i>struct</i> : struct)	
<b>Parameters</b>	<i>file-name</i>	The name of the file in which to write the struct.
	<i>struct</i>	The name of the struct instance to write to the file.

This recursively writes the contents of the *struct* to the *file-name* in binary format. If the struct contains other structs, those structs are also written to the file. If the file already exists, it is overwritten.

Syntax example:

```
files.write_binary_struct("b_file.dat", b_str)
```

## 31. Reflection API

This clause explains how to access structural information about the type system through the application programming interface (API), also known as the reflection API, and defines the reflection API itself. See also [Annex E](#) (for examples).

### 31.1 Introduction

*Reflection* (sometimes called *introspection*) is a programmatic interface into the meta-data of a program. Most object-oriented (OO) languages and systems supply some way of referring to meta-level entities—mainly type-related, such as classes, methods, and fields. The richness of modeling concepts supported by the *e* language calls for a much more comprehensive reflection facility than that of other languages. Reflection interfaces are typically used for constructing external developer aid tools such as class browsers, data browsers, debugging aids, source browsers, etc. They can also be used to implement generic utilities, where the modeling powers of the language such as inheritance and polymorphism are not strong enough, e.g., object serialization utilities like packing or register filling. The reflection interface in *e* is designed to facilitate such third-party applications and tools, where these problems typically arise.

#### 31.1.1 Representation

The *e* reflection facility is a class API. Each structural element in the program is represented by an object (an *e* struct instance). One object represents, for example, the type **int**, another represents the struct type **any\_struct**, another represents the method **to\_string()**, and so on. These representations are called *meta-objects*. Meta-objects are classified into different groups that are called *meta-types*. These *e* struct types form a hierarchy of abstractions and show the different relations between such entities. All of these meta-types have a common prefix to their name, **rf\_**.

#### 31.1.2 Structure

The interface is divided into three main parts: *type information*, *aspect information*, and *value query and manipulation*, which correspond to [31.2](#), [31.3](#), and [31.4](#) respectively. This grouping of the API's functionality cuts across meta-types so that the interface of one struct may consist of methods that are defined in different parts (e.g., some methods of the struct **rf\_event** are described in [31.2](#), others in [31.3](#), and the rest in [31.4](#)).

Each meta-type is introduced separately. Its location in the type hierarchy is denoted by showing its *like* inheritance (if any), which has the usual inheritance implications (e.g., the method **is\_private()**, which is defined for the struct **rf\_struct\_member**, also exists for **rf\_field**, which is *like* **rf\_struct\_member**). The concept behind each meta-type is explained, its methods are detailed, and each method's return type is set off by a colon (:). For example, in **rf\_named\_entity.get\_name()**: string, the type returned is a string.

#### 31.1.3 Terminology and conventions

- There is a possibility of confusion when dealing with meta-data and meta-types, as objects are used to represent types, methods, and so on; while they themselves, like any other object, are instances of types and have methods. However, for the sake of readability, and where there is no ambiguity, the phrase “the type/method/field” is shorthand for “an object representing the type/method/field.” For example, the method **rf\_struct.get\_methods()** returns a list of methods of this struct, which means it returns a list of objects representing this struct's methods.

Clarifying the concept of **like** and **when** inheritance (see [6.1](#)) is a major concern. Therefore, two trivial examples are used in many places to illustrate these definitions. One is the hierarchy of *dog*, with *bulldog* and *poodle* as its **like** heirs. The other is the struct **packet**, having an enumerated field **size**, and a

Boolean field `corrupt`, which allow for **when** subtypes such as `small packet` or `big corrupt packet`.

## 31.2 Type information

The core of the reflection API is the representation of types in *e*. This part of the interface enables all type-related queries concerning scalar types, list types, struct types, methods, fields, and events. From the viewpoint of the reflection API, units are simply structs (see [31.2.2](#) for how to query if a struct is a unit).

### 31.2.1 Named entities

This subclause defines the types of named entities.

#### 31.2.1.1 `rf_named_entity`

Named entities are types and struct members. Both kinds are entities that, once declared, become part of the lexicon of the language. All named entities have a name (string) and are either visible or hidden. The importance of this abstraction is related to [31.3](#).

- a) **`rf_named_entity.get_name()`**: string  
Returns the name of this entity.
- b) **`rf_named_entity.is_visible()`**: bool  
Returns TRUE if this entity is visible. Otherwise, returns FALSE. *Invisible (hidden) named entities* include fields and methods of any user-defined structs, which are not shown in printing and visualization tools.

The following method of `rf_named_entity` is described in section [31.3.1.3](#):

**`rf_named_entity.get_declaration()`**

#### 31.2.1.2 `rf_type`

This struct *like*-inherits from `rf_named_entity` (see [31.2.1.1](#)).

- a) **`rf_type.is_public()`**: bool  
Returns TRUE if this type has unrestricted access. Otherwise, returns FALSE (when this type was declared with a *package* modifier).
- b) **`rf_type.get_qualified_name()`**: string  
Returns the fully qualified name of the type (i.e., with the declaring package name followed by the `::` operator).

The following methods of `rf_type` are described in section [31.4.2](#):

**`rf_type.create_holder()`**  
**`rf_type.value_is_equal()`**  
**`rf_type.value_to_string()`**

### 31.2.2 Struct types: `rf_struct`

This struct *like*-inherits from `rf_type` (see [31.2.1.2](#)). See also [Clause 6](#).

- a) **`rf_struct.get_fields()`**: list of `rf_field`  
Returns a list containing all fields of this struct (declared by it or inherited from its parent type).

- b) **rf\_struct.get\_declared\_fields()**: list of rf\_field  
Returns a list containing all fields declared in the context of this struct (a subset of **rf\_struct.get\_fields()**).
- c) **rf\_struct.get\_field(name: string)**: rf\_field  
Returns the field of this struct with the *name* or NULL if no such field exists. Field names are unique in the context of a struct.
- d) **rf\_struct.get\_methods()**: list of rf\_method  
Returns a list containing all methods of this struct (declared by it or inherited from its parent type).
- e) **rf\_struct.get\_declared\_methods()**: list of rf\_method  
Returns a list containing all methods declared in the context of this struct (a subset of **rf\_struct.get\_methods()**). Methods that are declared by a parent type and extended or overridden in the context of this struct are not returned (see also [6.3](#)).
- f) **rf\_struct.get\_method(name: string)**: rf\_method  
Returns the method of this struct with the *name* or NULL if no such method exists. Method names are unique in the context of a struct.
- g) **rf\_struct.get\_events()**: list of rf\_event  
Returns a list of all events of this struct (declared by it or inherited from its parent type).
- h) **rf\_struct.get\_declared\_events()**: list of rf\_event  
Returns a list of all events declared in the context of this struct (a subset of **rf\_struct.get\_events()**). Events that are declared by a parent type and overridden in the context of this struct are not returned (see also [6.3](#)).
- i) **rf\_struct.get\_event(name: string)**: rf\_event  
Returns the event of this struct with the *name* or NULL if no such event exists. Event names are unique in the context of a struct.
- j) **rf\_struct.is\_unit()**: bool  
Returns TRUE if this struct is a unit. Otherwise, returns FALSE. Returning TRUE is the only indication this meta-object represents a unit rather than a regular struct type.

The following methods of **rf\_struct** are described in section [31.2.4.3](#):

```
rf_struct.is_contained_in()
rf_struct.is_disjoint()
rf_struct.is_independent()
rf_struct.get_when_base()
```

### 31.2.3 Struct members

Struct members are represented by instances of the meta-types **rf\_field** (see [31.2.3.2](#)), **rf\_method** (see [31.2.3.3](#)), and **rf\_event** (see [31.2.3.5](#)). Each member is introduced for the first time in the context of some struct—its *declaring struct*. Its access rights—one of **package**, **protected**, **private**, or **public** (the default)—are assigned to it upon its declaration.

#### 31.2.3.1 rf\_struct\_member

This struct *like*-inherits from **rf\_named\_entity** (see [31.2.1.1](#)).

- a) **rf\_struct\_member.get\_declaring\_struct()**: rf\_struct  
Returns the struct where this member was introduced. This applies also to the empty definition of methods or declarations of undefined methods.

- b) **rf\_struct\_member.applies\_to(*rf\_struct*)**: bool  
Returns TRUE if this struct member applies to instances of *rf\_struct*; i.e., this was declared by *rf\_struct* or by a different struct that contains *rf\_struct* (see also [6.2](#)). Otherwise, returns FALSE.
- c) **rf\_struct\_member.is\_private()**: bool  
Returns TRUE if this struct member was declared with the **private** access modifier; i.e., it is accessible only within the context of both its package and its declaring struct or its subtypes. Otherwise, returns FALSE.
- d) **rf\_struct\_member.is\_protected()**: bool  
Returns TRUE if this struct member was declared with **protected** access modifier; i.e., it is accessible only within the context of the declaring struct or its subtypes. Otherwise, returns FALSE.
- e) **rf\_struct\_member.is\_package\_private()**: bool  
Returns TRUE if this struct member was declared with **package** access modifier; i.e., it is accessible only within the context of the package where it was declared. Otherwise, returns FALSE.
- f) **rf\_struct\_member.is\_public()**: bool  
Returns TRUE if this struct member was declared without an access modifier; i.e., its access is not restricted. Otherwise, returns FALSE.

### 31.2.3.2 rf\_field

This struct *like*-inherits from **rf\_struct\_member** (see [31.2.3.1](#)).

- a) **rf\_field.get\_type()**: rf\_type  
Returns the declared type of the field.
- b) **rf\_field.is\_physical()**: bool  
Returns TRUE if the field is declared physical (i.e., with the % modifier (see [6.8](#))). Otherwise, returns FALSE. Physical fields are those that are packed when the struct is packed.
- c) **rf\_field.is\_ungenerated()**: bool  
Returns TRUE if the field is declared as ungenerated (i.e., with the ! modifier (see [6.8](#))). Otherwise, returns FALSE. Ungenerated fields are not generated automatically when the struct is generated.
- d) **rf\_field.is\_unit\_instance()**: bool  
Returns TRUE if the field is an instance of a unit (i.e., declared as **is instance** (see [7.2.2](#))). Otherwise, returns FALSE.

The following methods of **rf\_field** are described in section [31.4.3](#):

```
rf_struct.get_value()
rf_struct.get_value_unsafe()
rf_struct.set_value()
rf_struct.set_value_unsafe()
```

### 31.2.3.3 rf\_method

This struct *like*-inherits from **rf\_struct\_member** (see [31.2.3.1](#)).

- a) **rf\_method.get\_result\_type()**: rf\_type  
Returns the object that represents the result type of this method or NULL if the method does not return any value.
- b) **rf\_method.get\_parameters()**: list of rf\_parameter  
Returns a list of formal parameters of this method. If the method has no parameters, the list is empty.
- c) **rf\_method.is\_tcm()**: bool



Returns TRUE if this method may consume time; i.e., it is declared as a TCM. Otherwise, returns FALSE.

- d) **rf\_method.is\_inline()**: bool

Returns TRUE if the method is declared as *inline* (see [18.1.1](#)). Otherwise, returns FALSE.

- e) **rf\_method.get\_sampling\_event()**: rf\_event

Returns the object that represents the default sampling event in case this method is a TCM or NULL otherwise.

The following methods of **rf\_method** are described in section [31.3.1.1](#):

**rf\_method.get\_layers()**

**rf\_method.get\_relevant\_layers()**

The following methods of **rf\_method** are described in section [31.4.3](#):

**rf\_method.invoke()**

**rf\_method.invoke\_unsafe()**

**rf\_method.start\_tcm()**

**rf\_method.start\_tcm\_unsafe()**

#### 31.2.3.4 rf\_parameter

- a) **rf\_parameter.get\_name()**: string

Returns the name given to this parameter in the declaration method.

- b) **rf\_parameter.get\_type()**: rf\_type

Returns the type of this parameter.

- c) **rf\_parameter.is\_by\_reference()**: bool

Returns TRUE if this parameter is passed by reference; i.e., it was declared using \* (see [18.3.1](#)). Otherwise, returns FALSE.

#### 31.2.3.5 rf\_event

This struct *like*-inherits from **rf\_struct\_member** (see [31.2.3.1](#)).

The following method of **rf\_event** is described in section [31.3.3.2](#):

**rf\_event.get\_layers()**

The following methods of **rf\_event** are described in section [31.4.3](#):

**rf\_event.is\_emitted()**

**rf\_event.is\_emitted\_unsafe()**

**rf\_event.emit()**

**rf\_event.emit\_unsafe()**

### 31.2.4 Inheritance and when subtypes

There are two mechanisms for subtyping in *e*. One is OO single inheritance (*like inheritance*), where a struct is declared as derived from another. The other (*when subtyping*) is closer to predicate classes, where a behavioral or structural feature of an object is determined by some state or attribute. In both cases, a new struct type is defined in terms of an existing one. But, the relations between the two kinds of struct types are

different and they are represented by different kinds of meta-objects. Thus, there are two kinds of struct types: **like** structs and **when** subtypes.

The two mechanisms, **like** and **when**, do not mix. *Like* inheritance lays the basic type hierarchy. Only the leaves of the hierarchy tree, the structs that have no *like* subtypes, can serve as a base for *when* proliferations. Each **when** variant is a different type, but unlike **like** structs, these types are not derived from each other and do not form a hierarchy. To mark this difference, the set of **when** subtypes is called a *when family* and the **like** struct that serves as the base for these proliferations is called a *when base*. Also, a **when** subtype can be determined by a field that is declared in the context of another **when** subtype; however, such subtypes are part of the same *when family* with the same *when base*.

See also [Clause 6](#).

### 31.2.4.1 Canonical names

Each value of an enumerated or a Boolean field of an object can serve as a determinant of its structure and behavior. A set of one or more field/values pairs (determinants) corresponds to a potential **when** subtype. One such type can have a number of names by which it is identified—using fully qualified determinants or without them (e.g., `big'size packet` versus `big packet`)—and in a different determinant order (e.g., `big corrupt packet` versus `corrupt big packet`). However, the method `get_name()` returns a *canonical name*, the fully qualified determinants in the reverse order of the declaration of the fields. For example, `TRUE'corrupt big'size packet` is a canonical name of one of `packet`'s subtypes, given that field `corrupt` was defined after field `size`.

### 31.2.4.2 Explicit and significant subtypes

A **when** variant of a struct is called an *explicit* subtype if it is explicitly given some distinctive structural or behavioral content by some **when** or **extend** constructs (see [6.6](#) and [6.3](#)). Subtypes can be true variants of a struct, i.e., have distinct content, even when they are not explicitly defined: they consist of the conjunction of two or more explicit subtypes. These are called *significant* subtypes. Significant subtypes are important because each object in the program has exactly one type that describes it exhaustively (see [31.4.2](#)).

For example, the struct `packet` with enumerated field `size` (`big` or `small`) and Boolean field `corrupt` has four possible **when** subtypes. If only `big packet` and `corrupt packet` are defined as explicit variants of `packet` (using the constructs `when big packet {...}` and `when corrupt packet {...}`), then only they are explicit subtypes. In this case, `corrupt big packet` is also a significant subtype, since it has some distinctive features. On the other hand, `small packet` is neither explicit nor significant, since instances of it are equivalent to instances of `packet`.

### 31.2.4.3 Generalized relationships

There are a number of generalized relations that apply to both **like** structs and **when** subtypes, such as containment and mutual exclusion. However, regular inheritance relations, e.g., whether a struct is a direct parent type or a direct subtype of another, are applicable only to **like** structs.

- a) **rf\_struct.is\_contained\_in**(*rf\_struct*): bool

Returns TRUE if every instance of this struct is an instance of *rf\_struct*. Otherwise, returns FALSE. For example, `bulldog` is contained in `dog`; whereas, `corrupt small packet` is contained in `small packet`, in `corrupt packet`, in `packet`, and in itself, but `small packet` is not contained in `corrupt packet`.

- b) **rf\_struct.is\_disjoint**(*rf\_struct*): bool

Returns TRUE if every instance of this struct is not an instance of *rf\_struct* and vice versa; i.e., the two types are mutually exclusive. Otherwise, returns FALSE. **like** structs are disjoint if they are not

identical and neither one is contained in the other, e.g., *bulldog* and *poodle* are disjoint, *big packet* and *small packet* are disjoint, but *big packet* and *corrupt packet* are not.

- c) **rf\_struct.is\_independent**(*rf\_struct*): bool

Returns TRUE if an instance of this struct is possibly, but not necessarily, an instance of *rf\_struct*. Otherwise, returns FALSE. This relation holds only between two **when** subtypes that are neither contained nor mutually exclusive. For example, *big packet* is independent of *corrupt packet*, but not of *corrupt small packet*.

- d) **rf\_struct.get\_when\_base**(): *rf\_like\_struct*

Returns the struct that is the base of the **when** struct family. This struct itself is returned if it is not a **when** subtype (regardless of whether it actually contains **when** subtypes).

#### 31.2.4.4 rf\_like\_struct

This struct *like*-inherits from **rf\_struct** (see [31.2.2](#)).

- a) **rf\_like\_struct.get\_supertype**(): *rf\_like\_struct*

Returns the immediate **like** parent type of this struct.

- b) **rf\_like\_struct.get\_direct\_like\_subtypes**(): list of *rf\_like\_struct*

Returns the set of immediate subtypes of this struct in the **like** struct hierarchy.

- c) **rf\_like\_struct.get\_all\_like\_subtypes**(): list of *rf\_like\_struct*

Returns the set of all subtypes of this struct in the **like** struct hierarchy.

- d) **rf\_like\_struct.get\_when\_subtypes**(): list of *rf\_when\_subtype*

Returns the set of all defined subtypes in the **when** struct family for this struct. If this struct is not a leaf in the *like* hierarchy (i.e., it has **like** subtypes), the method returns an empty list. Any subtypes that are significant, but not defined, are not returned (see [31.2.4.5](#)).

The following method of **rf\_like\_struct** is described in section [31.3.2.3](#):

**rf\_like\_struct.get\_layers**()

#### 31.2.4.5 rf\_when\_subtype

This struct *like*-inherits from **rf\_struct** (see [31.2.2](#)).

- a) **rf\_when\_subtype.get\_short\_name**(): string

Returns a short version of the canonical name, i.e., without determinant qualification unless ambiguity requires it. For example, a type whose canonical name is *corrupt TRUE big size packet* would (normally) have the short name *corrupt big packet*. *Qualified determinants* appear in the short name when the same value name is a possible value of more than one field.

- b) **rf\_when\_subtype.is\_explicit**(): bool

Returns TRUE if this **when** subtype is explicitly defined in the program (by a **when** or an **extend** construct). Otherwise, returns FALSE (for both *significant* and *insignificant* subtypes).

- c) **rf\_when\_subtype.get\_contributors**(): list of *rf\_when\_subtype*

Returns the set of subtypes that contribute to the definition of this subtype, i.e., all the explicit subtypes where this subtype is contained, including itself if that case is explicitly defined.

### 31.2.5 List types

This subclause defines the list types.

### 31.2.5.1 **rf\_list**

This struct *like*-inherits from **rf\_type** (see [31.2.1.2](#)).

Lists are multi-purpose containers in *e*. The different list types are all instances of a generic definition similar to a parameterized type (template) in other OO languages. Any non-list type (scalars, strings, or structs) can serve as the element type of a list.

- a) **rf\_list.get\_element\_type()**: rf\_type  
Returns the element type of this list, e.g., the element type of `list of big packet` is `big packet`.
- b) **rf\_list.is\_packed()**: bool  
Return TRUE if this list is *packed* (a list with an element type whose size in bits is 16 or less). Otherwise, returns FALSE.

### 31.2.5.2 **rf\_keyed\_list**

This struct *like*-inherits from **rf\_list** (see [31.2.5.1](#)).

- rf\_keyed\_list.get\_key\_field()**: rf\_field  
Returns the field by which the list is mapped or NULL if the key is the object itself (i.e., when the list is defined as `(key: it)`).

## 31.2.6 Scalar types

Scalars in *e* have value semantics in assignment, parameter passing, equivalence, operators, etc. They are either enumerated, numeric, or Boolean types.

### 31.2.6.1 **rf\_scalar**

This struct *like*-inherits from **rf\_type** (see [31.2.1.2](#)).

- a) **rf\_scalar.get\_size\_in\_bits()**: int  
Returns the size of this scalar type in bits.
- b) **rf\_scalar.get\_range\_string()**: string  
Returns a string representation of the scalar range of values in the format of range modifiers (e.g., the string `"[ 1 . . 4 , 7 , 9 . . 10 ]"`).

### 31.2.6.2 **rf\_numeric**

This struct *like*-inherits from **rf\_scalar** (see [31.2.6.1](#)).

- rf\_numeric.is\_signed()**: bool  
Returns TRUE if the numeric type is signed. Otherwise, returns FALSE.

### 31.2.6.3 **rf\_enum**

This struct *like*-inherits from **rf\_scalar** (see [31.2.6.1](#)).

- a) **rf\_enum.get\_items()**: list of rf\_enum\_item  
Returns the set of named values for this type. The legal values of an **enum** type (see [4.3.2.3](#)) are not restricted by a range declaration, e.g., the type introduced by the statement `type my_color: color [red..blue]` has the same items as `type color`. Such declarations only effect generation properties of the type.
- b) **rf\_enum.get\_item\_by\_value(value: int)**: rf\_enum\_item  
Returns the named value object for *value* or NULL if no such value exists in this type's range.
- c) **rf\_enum.get\_item\_by\_name(name: string)**: rf\_enum\_item  
Returns the named value object for *name* or NULL if no value by such name exists in this type's range.

The following method of **rf\_enum** is described in section [31.3.2.2](#):

**rf\_enum.get\_layers()**

#### 31.2.6.4 rf\_enum\_item

Enum items are pairs of identifier-integer, which are the possible values of a variable of that **enum** type. The integer values of **enum** items are the numbers assigned to them explicitly in the declaration (e.g., [`red = 3, green = 17`]) or the default (consecutive) numbers.

- a) **rf\_enum\_item.get\_name()**: string  
Returns the identifier associated with this item as a string.
- b) **rf\_enum\_item.get\_value()**: int  
Returns the integer value associated with this item as a signed integer.

#### 31.2.6.5 rf\_bool

This struct *like*-inherits from **rf\_scalar** (see [31.2.6.1](#)).

Boolean types in *e* are the predefined type **bool** and its (possibly user-defined) width derivatives such as `bool (bits:8)`. Boolean types have no special features others than those declared for **rf\_scalar**.

#### 31.2.6.6 rf\_string

This struct *like*-inherits from **rf\_type** (see [31.2.1.2](#)).

Strings in *e* are instances of a special built-in type, which is neither scalar nor compound (a struct or list). Unlike the other three types (enumerated, numeric, or Boolean), there is only one string type; thus, the meta-type **rf\_string** is a singleton. It does not have any special features other than those declared for **rf\_type**.

### 31.2.7 Port types

Ports in *e* are special purpose objects that serve to bind different units in the verification environment and specifically to interconnect with the DUT. Each port is an instance of one the four port types. There are three kinds of parameterized port types: `simple_port`, `buffer_port`, and `method_port`, and a non-parameterized kind: `event_port`.

#### 31.2.7.1 rf\_port

This struct *like*-inherits from **rf\_type** (see [31.2.1.2](#)).

- a) **rf\_port.is\_input()**: bool

Returns TRUE if this port type is declared as an input with the **in** or the **inout** specifier.

- b) **rf\_port.is\_output()**: bool

Returns TRUE if this port type is declared as an output with the **out** or the **inout** specifier.

### 31.2.7.2 rf\_simple\_port

This struct *like*-inherits from **rf\_port** (see [31.2.7.1](#)).

**rf\_simple\_port.get\_element\_type()**: rf\_type

Returns the element type of this port type.

### 31.2.7.3 rf\_buffer\_port

This struct *like*-inherits from **rf\_port** (see [31.2.7.1](#)).

**rf\_buffer\_port.get\_element\_type()**: rf\_type

Returns the element type of this port type.

### 31.2.7.4 rf\_event\_port

This struct *like*-inherits from **rf\_port** (see [31.2.7.1](#)).

### 31.2.7.5 rf\_method\_port

This struct *like*-inherits from **rf\_port** (see [31.2.7.1](#)).

**rf\_method\_port.get\_element\_type()**: rf\_type

Returns the method type of this method port.

## 31.2.8 Querying for types: rf\_manager

The starting point in every query into the type information is a set of services that are not related to any specific kind of meta-objects. They are scoped together as methods of a singleton class named **rf\_manager**, the instance of which is under **global**. This struct has other general services that are defined in [31.3.5](#), [31.4.1](#), and [k](#).

- a) **rf\_manager.get\_type\_by\_name(name: string)**: rf\_type

Returns the type with *name* or NULL if no type by that name exists in the system.

- b) **rf\_manager.get\_user\_types()**: list of rf\_type

Returns a list of all the types declared in the user modules.

The following method of **rf\_manager** is described in section [31.3.5](#):

**rf\_manager.get\_module\_by\_name()**

**rf\_manager.get\_module\_by\_index()**

**rf\_manager.get\_user\_modules()**

**rf\_manager.get\_package\_by\_name()**

The following method of **rf\_manager** is described in section [31.4.1](#):

**rf\_manager.get\_struct\_of\_instance()**

**rf\_manager.get\_exact\_subtype\_of\_instance()**

The following method of **rf\_manager** is described in section [31.4.4](#):

```

rf_manager.get_list_element()
rf_manager.get_list_element_unsafe()
rf_manager.set_list_element()
rf_manager.set_list_element_unsafe()
rf_manager.get_list_size()
rf_manager.get_list_size_unsafe()

```

### 31.3 Aspect information

The structure and behavior of objects at runtime consists of fields, methods, events, etc. In *e*, the definition of these constituents can be separated into different modules of the software and extended on a per-type basis as part of the aspect-oriented modeling paradigm (i.e., decomposed as different concerns). Thus, the mapping between how the code is laid out and imported, and the end-result of accumulated software layers once all of the extensions have been resolved, is non-trivial.

This part of the API primarily models the mapping between named entities and the structure of their definitions in the source code. Different meta-objects are used to represent the elements in the code that constitute the definition of a given named entity; they are classified according to the kind of named entity they define. **rf\_definition\_element** serves as a common base type for these types (see [31.3.1.3](#)).

#### 31.3.1 Definition elements

This subclause describes the definition elements.

##### 31.3.1.1 Extensible entities and layers

In common OO languages, the definition of a class begins and ends in one single stretch of code. Conversely the definition of structs in *e* can be separated between different locations in the source files. It is introduced and initially defined by a **struct** statement and then possibly further defined by later **extend** statements. Each such “piece” of definition is called a *struct layer*. An enumerated type can similarly be initially defined with some set of named values and extended later with more named values. Each of these is an *enum layer*.

Methods can be overridden or refined not only in subtypes, but also later in the same struct [by **is also/first/only** constructs (see [18.1.3](#))]. Thus, the definition of a method for some given object is a series of one or more definition “pieces” which are called *method layers*. Events, like methods, are declared once in some struct and are possibly overridden later in the same struct or in subtypes. These concepts are explained as follows (see [31.3.3](#)).

Generally speaking, entities that can be declared at one location in the source code and extended in later locations, such as struct types, enum types, methods and events, are called *extensible entities*. The definition of *extensible entities* consists of a series of one or more elements (layers), the first of which is the *declaration* and the rest are *extensions*. Named entities of all kinds can be queried for their declaration (see [31.3.1.3](#)). Extensible named entities (e.g., **rf\_struct** and **rf\_method**) can also be queried for their extensions (e.g., see [31.3.2.3](#)).

##### 31.3.1.2 Anomalies of definition elements

The separation between a named entity and its definition is natural where extensible entities are concerned. However, it is somewhat artificial for non-extensible entities, e.g., numeric types and fields. Nevertheless, the same scheme applies trivially to non-extensible entities. Their definition consists of exactly one element—the *declaration*. For example, the **rf\_field** object (see [31.4.3](#)) that represents the field `size` of the

struct `packet` can be queried for the source location of its declaration, not directly, but through a different object (of type `rf_definition_element` (see [31.3.1.3](#)), which represents its declaration.

Moreover, some named entities are not explicitly defined by *e* code at all and so have no definition elements whatsoever, not even a declaration. For example, list types are instantiations of a parameterized built-in type. They are used in *e* code and represented in the type system just as any other type, but they are never defined by *e* code itself. See also [31.3.2](#).

### 31.3.1.3 `rf_definition_element`

- a) **`rf_definition_element.get_defined_entity()`**: `rf_named_entity`  
Returns the named entity that is being defined by this definition element; i.e., this element is part of the definition of the returned named entity.
- b) **`rf_definition_element.get_module()`**: `rf_module`  
Returns the module where this definition element appears.
- c) **`rf_definition_element.get_source_line_no()`**: `int`  
Returns the line number of the beginning of the clause in the source file.
- d) **`rf_definition_element.is_before(rf_definition_element)`**: `bool`  
Returns `TRUE` if this definition element appears before *rf\_definition\_element* in the load order. Otherwise, returns `FALSE`. This is based on a *full-order relation* on definition elements, which is defined as the ordinal number of modules and then the line number in the file.
- e) **`rf_definition_element.get_documentation()`**: `string`  
Returns the inline documentation of this definition element. *Inline documentation* is the comment in the consecutive lines directly preceding the definition in the source files. An empty string is returned if the source file is not found.
- f) **`rf_definition_element.get_documentation_lines()`**: `list of string`  
Returns the inline documentation of this definition element as a list of strings separated by newline characters in the source file. An empty list is returned if the source file is not found.
- g) **`rf_named_entity.get_declaration()`**: `rf_definition_element`  
Returns the declaration (the first definition element) of this entity or `NULL` for any types defined implicitly as variants of existing types (see [31.3.2](#)).

### 31.3.2 Type layers

**enum** and struct types are extensible entities, so their definition can consist of one or more layers. Other kinds of types are not extensible and so have only the declaring layer. However, not all types have explicit definitions. Some of these types can be used in context, without being previously declared:

- Numeric types can be used in context with a size modification (e.g., `uint (bits: 16)`). The size modification implies a different type, but one which has no explicit declaration.
- **enum** types can be spelled out inline, they have no separate declarations or explicit names.
- List types are instances of predefined parameterized types; they are not declared or defined in *e*.
- Not all **when** subtypes are explicitly defined, but they can still be used in context as types.

The first two cases are scalar types that could have been declared and given an explicit name by a **type** statement. In the other two cases, there is no way to make the type declaration explicit. Some **when** subtypes are explicitly defined in a different sense by using **when** or **extend** constructs. Even then, the layers of the **when** subtypes cannot always be separated from those of other subtypes or the **when** base. From the viewpoint of aspect information, all these cases are treated in the same way: All types that fall under one of the previous cases do not have any layers. Calling the method `get_declaration()` (see [31.3.1](#)) on them



returns `NULL`, and for implicitly defined **enum**, calling `get_layers()` (see [31.3.2.1](#)) returns an empty list. As for structs, the service `get_layers()` is restricted to **like** structs.

### 31.3.2.1 rf\_type\_layer

This struct *like*-inherits from **rf\_definition\_element** (see [31.3.1.3](#)).

Structs and **enums** are extensible entities; they are defined in layers. Struct and enum layers are both type layers. This abstraction does not have features of its own, but is used by other services (see `get_type_layers()` in [31.3.4](#)).

### 31.3.2.2 rf\_enum\_layer

This struct *like*-inherits from **rf\_type\_layer** (see [31.3.2.1](#)).

- a) **rf\_enum\_layer.get\_added\_items()**: list of `rf_enum_item`  
Returns the named values added by this enum layer.
- b) **rf\_enum\_layer.get\_layers()**: list of `rf_enum_layer`  
Returns all enum layers that constitute this enum type.

### 31.3.2.3 rf\_struct\_layer

This struct *like*-inherits from **rf\_type\_layer** (see [31.3.2.1](#)).

- a) **rf\_struct\_layer.get\_field\_declarations()**: list of `rf_definition_element`  
Returns the field declarations added to the struct by this struct layer.
- b) **rf\_struct\_layer.get\_method\_layers()**: list of `rf_method_layer`  
Returns the method layers added to the struct by this struct layer.
- c) **rf\_struct\_layer.get\_event\_layers()**: list of `rf_event_layer`  
Returns the event layers added to the struct by this struct layer.
- d) **rf\_like\_struct.get\_layers()**: list of `rf_struct_layer`  
Returns all struct layers that constitute this struct type.

## 31.3.3 Method and event layers

Once a method in *e* is declared for a given struct, it can never be replaced by a different method in a subtype or a later extension. Rather, all later modifications of the definition, in all three modes, **also**, **first**, and **only**, in extensions as well as in *when* subtypes and *like* heirs, are definition elements of the same method—they are *method layers*. For example, the method `bark()`, once declared for struct `dog`, is one and the same for all kinds of dogs. But different method layers may be executed upon calling `bark()` for different dog objects, so they display different behaviors.

The reason for this deviation from standard OO terminology is *e* can be used to modify the behavior in derived structs, as well as when variants, and in later extensions of that same struct. Therefore, the need to distinguish between the method (the common semantics or message) on the one side and the definition of the behavior associated with it for some set of objects on the other side is more acute. These same considerations and terminology also apply to *events*.

### 31.3.3.1 rf\_method\_layer

This struct *like*-inherits from **rf\_definition\_element** (see [31.3.1.3](#)).

- a) **rf\_method\_layer.get\_context\_layer()**: `rf_struct_layer`

Returns the struct layer where this method layer appears.

- b) **type rf\_extension\_mode**: [empty, undefined, is, also, first, only]  
**rf\_method\_layer.get\_extension\_mode()**: rf\_extension\_mode

Returns one of the values—empty, undefined, is, also, first, or only—according to how this method layer was declared.

- c) **rf\_method.get\_layers()**: list of rf\_method\_layer

Returns a list of all layers of this method in all struct types where it is defined. The returned list is ordered by load order from early to late.

- d) **rf\_method.get\_relevant\_layers(rf\_struct)**: list of rf\_method\_layer

Returns a list of all layers of this method that apply to *rf\_struct*. If *rf\_struct* does not have this method at all, an empty list is returned. For example, the method **to\_string()** (see [28.4.4](#)) is defined for every struct in *e*, so calling **get\_layers()** returns all extensions of this method in the system. However, calling **get\_relevant\_layer()** for the struct *packet* only returns the extensions of **to\_string()** defined in the context of the struct *packet* and its subtypes.

### 31.3.3.2 rf\_event\_layer

This struct *like*-inherits from **rf\_definition\_element** (see [31.3.1.3](#)).

- a) **rf\_event\_layer.get\_context\_layer()**: rf\_struct\_layer

Returns the struct layer where this event layer appears.

- b) **rf\_event\_layer.get\_extension\_mode()**: rf\_extension\_mode

Returns either *is* or *only*, according to how this event layer was declared (see **type rf\_extension\_mode** in [31.3.3.1](#)). Other extension modes, such as *first* or *also*, are not applicable to events.

- c) **rf\_event.get\_layers()**: list of rf\_event\_layer

Returns a list of all layers of this event in all struct types where it is defined. The returned list is ordered by load order from early to late.

## 31.3.4 Modules and packages

This subclause describes the modules and packages.

### 31.3.4.1 rf\_module

*Modules* are simply *e* files. However, with the ability to extend structs and separate different concerns or crosscuts of a system, modules play an important role in organizing the program. If a struct may be considered the vertical encapsulation principle, then modules are the horizontal one. A struct consists of a number of related layers of definition in different modules and, symmetrically, the module consists of a number of related layers of different structs—it can be thought of as a layer of the entire system. Thus, modules can be queried for their overall contribution to the structure of a system in the reflection API.

- a) **rf\_module.get\_name()**: string

Returns the name of this module, basically the name of the *e* file without the *.e* extension.

- b) **rf\_module.get\_index()**: int

Returns this module's ordinal number in the load order.

- c) **rf\_module.get\_type\_layers()**: list of rf\_type\_layer

Returns a list of all the type layers defined in this module. A module's overall contribution to the structure of a system is the set of declarations of new types and extensions of existing types.

- d) **rf\_module.get\_package()**: rf\_package

Returns the *e* package with which this module is associated. Any modules that are not explicitly associated with some package [using the **package** statement (see [23.1](#))] are implicitly part of the package **main**.

- e) **rf\_module.is\_user\_module()**: bool

Returns TRUE if the module is user-defined. Otherwise, returns FALSE.

- f) **rf\_module.is\_encrypted()**: bool

Returns TRUE if the module is encrypted (see [Clause 33](#)). Otherwise, returns FALSE.

#### 31.3.4.2 rf\_package

A *package* (see [Clause 23](#)) is a set of one or more *e* modules that together implement some closely related functionality. This package defines a scope for restricting the access of named entities. It also is represented in the reflection API by a meta-object.

- a) **rf\_package.get\_name()**: string

Returns the name of this package.

- b) **rf\_package.get\_modules()**: list of rf\_module

Returns the set of modules associated with this package.

#### 31.3.5 Querying for aspects

Similar to the services for type information queries (see [31.2.8](#)), the following services can be used to perform aspect information queries.

- a) **rf\_manager.get\_module\_by\_name(name: string)**: rf\_module

Returns the module with the *name* or NULL if no module by this name is currently loaded.

- b) **rf\_manager.get\_module\_by\_index(index: int)**: rf\_module

Returns the module with the given *index* in the load order.

- c) **rf\_manager.get\_user\_modules()**: list of rf\_module

Returns a list of all user modules that are currently loaded.

- d) **rf\_manager.get\_package\_by\_name(name: string)**: rf\_package

Returns the package with the *name* or NULL if no package by this name is currently loaded.

### 31.4 Value query and manipulation

The parts of the API described in previous subclauses, type information and aspect information, both reflect static features of the program. During a run of the program, values are being manipulated. Each of those values is an instance of a type and all operations carried upon them are defined by their type. This part of the API enables the user to query and manipulate values using the representations of types. This feature is known as *meta-programming*. It can serve to construct data browsers, debugging aids, and other generic runtime features.

See also [5.2](#) and [5.8.2](#).

#### 31.4.1 Types of objects

The natural entry point for querying or manipulating objects is getting a representation of their type. For any given object, there is always one most specific type of which it is an instance, even if that type has not been explicitly defined in the code (i.e., it is a cross of a number of explicitly defined **when** subtypes). A query can be generated for the **like** struct of an instance and any **when** variants discarded, or the query can be for

the specific **when** subtype. The **when** subtype of an instance depends on its state, which may change with the course of the run.

- a) **rf\_manager.get\_struct\_of\_instance**(*instance*: base\_struct): rf\_like\_struct  
Returns the most specific **like** struct of the struct *instance* and disregards any **when** variants, even if they apply to the instance. To query for the specific **when** subtype of an object, use: **get\_exact\_subtype\_of\_instance**().
- b) **rf\_manager.get\_exact\_subtype\_of\_instance**(*instance*: base\_struct): rf\_struct  
Returns the type of *instance*. The returned meta-object represents the most specific significant type that applies to the *instance*, i.e., the one containing all other types that apply to the instance. For example, if the parameter is a `packet`, which has the defined subtypes `big packet` and `corrupt packet`, and a particular `packet` happens to be both `corrupt` and `big`, then the returned type would be `corrupt big packet`, even though it is not a *defined* subtype.  
  
The static type of a field is sometimes more specific than the exact subtype of the object that is the field's actual value: This happens when the static type is an insignificant **when** subtype (see [31.2.4.2](#)).

### 31.4.2 Values and value holders

When dealing with values in a generic way (meta-programming), there needs to be some safe way to refer to values of all types: struct instances, lists, strings, and scalars. Since these values are very different in their semantics and there is no abstract type common to all, the reflection API wraps values of all types with an object called **rf\_value\_holder**. This object holds a value together with its type, and guarantees its consistency and continuity when the original variable goes out of scope, and across garbage collections.

Value holders are returned from value queries or explicitly created by the user. They are used in setting values or calling methods. Actual uses of the value itself, however, involve passing through an *untyped* value and brute casting, which is not type-safe. Two operators are implemented generically for all values—equating and getting a string representation.

- a) **rf\_value\_holder.get\_type**(): rf\_type  
Returns the type of this value. When the value is a struct instance, the type of the holder is not necessarily the most specific subtype of that instance (e.g., a legal value holder whose type is **any\_struct** can hold an instance of `packet`).
- b) **rf\_value\_holder.get\_value**(): untyped  
Enables [by using the **unsafe** operator (see [5.8.2](#))] assignment of the value into a typed variable. The variable shall be a type to which this value is assignable according to *e* casting rules; however, this cannot be enforced.
- c) **rf\_type.create\_holder**(*value*: untyped): rf\_value\_holder  
Returns a value holder of this type for *value*, which shall be an instance of this type (or of a subtype in case it is a struct type). Very simple sanity checks are performed on the value; if they fail, an exception is thrown. These checks are by no means exhaustive; it is the user's responsibility to create the right holder for a value.
- d) **rf\_type.value\_is\_equal**(*value1*: untyped, *value2*: untyped): bool  
Returns TRUE if *value1* and *value2* are equivalent or identical [using the same semantics as that of the `==` operator (see [4.10.2](#))]. Otherwise, returns FALSE. The behavior is not defined if one of the two values is not of this type.
- e) **rf\_type.value\_to\_string**(*value*: untyped): string  
Returns a string representation of *value* (this is the same as the **to\_string**() operator). The behavior is not defined if the value is not of this type.

### 31.4.3 Object operators

Object operators include reading and writing fields, calling methods, and emitting or monitoring events. These operators are available with meta-objects that represent struct members. Value holders are the safe way to handle values in a generic way (see [31.4.2](#)). However, using them involves the dynamic allocation of memory, which can impact performance where this feature is heavily used.

Each object operator has two versions: One uses value holders and makes some checks, throwing exceptions in the cases where preconditions do not hold. The other uses bare *untyped* values (see [5.2](#)) and skips checks. This brute force version of each operator is marked as `unsafe` and should be avoided where possible.

- a) **`rf_field.get_value(instance: base_struct): rf_value_holder`**  
Returns the value of this field in the struct *instance*. If the struct type of the instance does not have this field, an exception is thrown.
- b) **`rf_field.get_value_unsafe(instance: base_struct): untyped`**  
Returns the value of this field in the struct *instance*. If the struct type of the instance does not have this field, the behavior is undefined.
- c) **`rf_field.set_value(instance: base_struct, value: rf_value_holder)`**  
Sets the value of this field in the struct *instance* to the new *value*. If the struct type of the instance does not have this field, an exception is thrown.
- d) **`rf_field.set_value_unsafe(instance: base_struct, value: untyped)`**  
Sets the value of this field in the struct *instance* to the new *value*. If the struct type of the instance does not have this field, the behavior is undefined.
- e) **`rf_method.invoke(instance: base_struct, parameters: list of rf_value_holder): rf_value_holder`**  
Calls this method on the struct *instance*, using the list of (zero or more) values as the method's *parameters*, and returns a value holder of the method's return value (or NULL if the method has none). If the struct type of the instance does not have this method or there is a mismatch in the number and types of parameters, an exception is thrown. This method cannot be called from an `rf_method` which is time consuming.
- f) **`rf_method.invoke_unsafe(instance: base_struct, parameters: list of untyped): untyped`**  
Calls this method on the struct *instance*, with the list of (zero or more) values as the method's *parameters*, and returns the method's return value. If this method does not return a value, the value returned from `invoke_unsafe` is undefined. If the struct type of the instance does not have this method or there is a mismatch in the number and types of parameters, the behavior is undefined. This method cannot be called from an `rf_method` which is time consuming.
- g) **`rf_event.is_emitted(instance: base_struct): bool`**  
Returns TRUE if this event of the *instance* was emitted so far in the current cycle. Otherwise, returns FALSE. If the struct type of the instance does not have this event, an exception is thrown.
- h) **`rf_event.is_emitted_unsafe(instance: base_struct): bool`**  
Returns TRUE if this event of the *instance* was emitted so far in the current cycle. Otherwise, returns FALSE. If the struct type of the instance does not have this event, the behavior is undefined.
- i) **`rf_event.emit(instance: base_struct)`**  
Emits the event on the *instance*. If the struct type of the instance does not have this event, an exception is thrown.
- j) **`rf_event.emit_unsafe(instance: base_struct)`**  
Emits the event on the *instance*. If the struct type of the instance does not have this event, the behavior is undefined.
- k) **`rf_method.start_tcm(instance: any_struct, parameters: list of rf_value_holder)`**

Starts this TCM on the *instance* given as a parameter. If the struct type of the instance does not declare this TCM, an error is issued. Similarly, if the given parameters are not of the types in the order required by this TCM, or this method is not a TCM, an error is issued. Note that this TCM may have a return value, but it is not accessible with **start\_tcm()**.

- l) **rf\_method.start\_tcm\_unsafe**(*instance*: any\_struct, *parameters*: list of untyped)

Starts this TCM on the instance given as a parameter. If the struct type of the instance does not declare this TCM, the behavior is undefined. Similarly, if the given parameters are not of the types in the order required by this TCM, or this method is not a TCM, the behavior is undefined. Note that this TCM may have a return value, but it is not accessible with **start\_tcm\_unsafe()**.

### 31.4.4 List operators

The three main list operators—reading an element, writing to an index, and querying the size—are available as general services, i.e., methods of **rf\_manager**. Two versions of the operators are available: the safe version, using value holders, and the brute one, using untyped values. See also [31.4.3](#).

- a) **rf\_manager.get\_list\_element**(*list*: rf\_value\_holder, *index*: int): rf\_value\_holder  
Returns the value of the given *list* at the given *index*. If the value is not a list or the index is out of bounds, an exception is thrown.
- b) **rf\_manager.get\_list\_element\_unsafe**(*list*: untyped, *index*: int): untyped  
Returns the value of the given *list* at the given *index*. If the value is not a list or the index is out of bounds, the behavior is undefined.
- c) **rf\_manager.set\_list\_element**(*list*: rf\_value\_holder, *index*: int, *new\_value*: rf\_value\_holder)  
Sets the value of the given *list* at the given *index*. If the first parameter is not a list, the index is out of bounds, or the new value is not of an instance of the list's element type, an exception is thrown.
- d) **rf\_manager.set\_list\_element\_unsafe**(*list*: untyped, *index*: int, *new\_value*: untyped)  
Sets the value of the given *list* at the given *index*. If the first parameter is not a list, the index is out of bounds, or the new value is not of an instance of the list's element type, the behavior is undefined.
- e) **rf\_manager.get\_list\_size**(*list*: rf\_value\_holder): int  
Returns the number of elements currently in the given *list*. If the value is not a list, an exception is thrown.
- f) **rf\_manager.get\_list\_size\_unsafe**(*list*: untyped): int  
Returns the number of elements currently in the given *list*. If the value is not a list, the behavior is undefined.

## 32. Predefined resource sharing control structs

This clause describes some predefined methods that are useful in controlling TCMs and resource sharing between TCMs. See also [Clause 28](#).

### 32.1 Semaphore methods

The *e* language provides three predefined structs that are useful in controlling resource sharing between TCMs, as follows:

— **semaphore**

This is the typical semaphore. The maximum value ranges between 1 and MAX\_INT. By default, it is MAX\_INT and the initial value (number of available resources) is 0.

— **rdv\_semaphore**

A rendezvous semaphore is a semaphore with no resources. It requires the meeting of a producer and a consumer for either to proceed. When they finally proceed, the **up()** thread always runs first, followed immediately by the **down()** thread.

— **locker**

The methods of this struct provide a fair, FIFO-ordered sharing of resources between multiple competing methods.

A *locker* is useful when a single entity needs to prevent others from a shared resource. **lock()** and **release()** need to be issued by the same entity. A *semaphore* can be more flexible.

[Table 44](#) gives a brief description of the predefined methods of the **semaphore** and **rdv\_semaphore** structs.

**Table 44—Semaphore methods**

Method	Description
<b>up()</b>	Increments the semaphore's value. Blocks if the value is already the maximum possible.
<b>down()</b>	Decrements the semaphore's value. Blocks if the value is already 0.
<b>try_up()</b>	Increments the semaphore's value. If the value is already the maximum possible, returns without blocking.
<b>try_down()</b>	Decrements the semaphore's value. If the value is already 0, returns without blocking.
<b>set_value()</b>	Sets the initial value of the semaphore.
<b>get_value()</b>	Returns the current value of the semaphore.
<b>set_max_value()</b>	Sets an upper limit to the possible value of the semaphore.
<b>get_max_value()</b>	Returns the maximum possible value.

[Table 45](#) describes the predefined methods of the **locker** struct.

**Table 45—Locker methods**

Method	Description
<b>lock()</b>	The first TCM to call the <b>lock()</b> method of a field of type <b>locker</b> gets the lock and can continue execution. The execution of the other TCMs is blocked.
<b>release()</b>	When a TCM that has the lock calls <b>release()</b> , control goes to the next TCM in line. The order in which the lock is granted is by a FIFO order of client <b>lock()</b> requests.

## 32.2 How to use the semaphore struct

A field of type **semaphore** typically serves as a synchronization object between two types of TCMs: producer and consumer.

- Any consumer TCM uses the predefined **down()** TCM of the semaphore to gain control of a new resource managed by the semaphore. If no resources are available at the time **down()** is called, the consumer TCM is blocked until such a resource is available.
- Any producer TCM uses the predefined **up()** TCM of the semaphore to increase the amount of available resources of the semaphore. This resource is made available for consumer TCMs. If the semaphore already contains the maximum number of resources at the time **up()** is called, the producer TCM is blocked until a semaphore resource is consumed.

The amount of available resources is zero (0) by default, but can be set otherwise by using the **set\_value()** method. The current amount of available resources can be obtained using the **get\_value()** method.

There is a limit to the possible number of available resources. Typically, the maximum is **MAX\_INT**, but it can be set to other values between 0 and **MAX\_INT** by using the **set\_max\_value()** method. The current limit for available resources can be obtained using the **get\_max\_value()** method.

Any producer TCM is blocked if the semaphore already holds the maximum number of available resources.

### 32.2.1 up() and down()

<b>Purpose</b>	Synchronize producer and consumer TCMs
<b>Category</b>	Predefined TCM of <b>semaphore</b> struct
<b>Syntax</b>	<i>semaphore.up()</i> <i>semaphore.down()</i>
<b>Parameters</b>	<i>semaphore</i> An expression of type <b>semaphore</b> or <b>rdv_semaphore</b> .

The **up()** TCM increases the number of available resources of the semaphore by 1. If the number of available resources is already the maximum, the TCM is blocked. Blocked calls to **up()** are serviced according to their request order (on a FIFO basis).

The **down()** TCM decreases the number of resources of the semaphore by 1. If no resources are available, the TCM is blocked. Blocked calls to **down()** are serviced according to their request order (on a FIFO basis).



In an **rdv\_semaphore**, **up()** and **down()** are blocked unless they coincide. The **down()** TCM always breaks the block first.

Syntax example:

```
sem1.up( );
sem1.down( )
```

### 32.2.2 try\_up() and try\_down()

<b>Purpose</b>	Synchronize producer and consumer methods
<b>Category</b>	Predefined method of <b>semaphore</b> struct
<b>Syntax</b>	<i>semaphore</i> . <b>try_up()</b> : bool <i>semaphore</i> . <b>try_down()</b> : bool
<b>Parameters</b>	<i>semaphore</i> An expression of type <b>semaphore</b> or <b>rdv_semaphore</b> .

The **try\_up()** and **try\_down()** methods try to increment or decrement the number of available resources by 1, respectively. If the number of available resources is already at its maximum or minimum, respectively, these methods return immediately without any effect (in particular, no blocking). If the number of resources was changed, the returned value is TRUE. If the number of resources was not changed, the returned value is FALSE.

- The FIFO order of service of the semaphore is kept even when the **try\_up()** and **try\_down()** methods are involved, e.g., a **try\_up()** shall never succeed if there are pending calls to **up()**.
- **try\_up()** and **try\_down()** never generate a context switch.

Syntax example:

```
compute sem1.try_up( );
compute sem1.try_down( )
```

### 32.2.3 set\_value() and get\_value()

<b>Purpose</b>	Set and get the number of available resources of a semaphore
<b>Category</b>	Predefined method of <b>semaphore</b> struct
<b>Syntax</b>	<i>semaphore</i> . <b>set_value</b> ( <i>new_value</i> : int) <i>semaphore</i> . <b>get_value</b> (): int
<b>Parameters</b>	<i>semaphore</i> An expression of type <b>semaphore</b> or <b>rdv_semaphore</b> .
	<i>new_value</i> An expression of type signed int.

The **set\_value()** method sets the number of available resources of the semaphore. By default, a semaphore is initialized with zero (0) available resources. The new value shall be a non-negative integer, no larger than MAX\_INT. If the **set\_max\_value()** method of the struct was used, the new value shall also be smaller or equal to the last setting of the maximum number of resources. If these conditions do not hold, a runtime error shall be issued.

- **set\_value()** cannot be called if either **up()** or **down()** was previously called. In such case, an error shall be issued. Setting the value of an **rdv\_semaphore** to something other than zero (0) shall also result in a runtime error.
- The **get\_value()** method returns the current number of available resources of the semaphore.

Syntax example:

```
sem1.set_value(7);
cur_value = sem1.get_value()
```

### 32.2.4 set\_max\_value() and get\_max\_value()

<b>Purpose</b>	Set and get the maximum number of available resources of a semaphore
<b>Category</b>	Predefined method of <b>semaphore</b> struct
<b>Syntax</b>	<i>semaphore.set_max_value(new_value: int)</i> <i>semaphore.get_max_value(): int</i>
<b>Parameters</b>	<i>semaphore</i> An expression of type <b>semaphore</b> or <b>rdv_semaphore</b> .
	<i>new_value</i> An expression of type signed int.

The **set\_max\_value()** method sets the maximum number of available resources of the semaphore. By default, a semaphore is initialized with a maximum of **MAX\_INT** available resources. The new value shall be a positive integer, no larger than **MAX\_INT**. If **set\_value()** was previously called, the new maximum shall not be smaller than the number of available resources. If these conditions do not hold, a runtime error shall be issued.

- The value of an **rdv\_semaphore** is constantly zero (0). Therefore its default maximum value is zero (0), and it cannot be set to a value other than that. Trying to do so shall result in a runtime error.
- **set\_max\_value()** cannot be called if either **up()** or **down()** was previously called. In such case, an error shall be issued.
- It is safer to invoke the **set\_max\_value()** method before any other semaphore method.
- The **get\_max\_value()** method returns the current limit for available resources of the semaphore.

Syntax example:

```
sem1.set_max_value(17);
cur_max_value = sem1.get_max_value()
```

### 32.2.5 lock() and release()

<b>Purpose</b>	Control access to a shared resource
<b>Category</b>	Predefined TCM of <b>locker</b> struct
<b>Syntax</b>	<i>locker-exp</i> . <b>lock()</b> <i>locker-exp</i> . <b>release()</b>
<b>Parameters</b>	<i>locker-exp</i> An expression of type <b>locker</b> .

**locker** is a predefined struct with two predefined methods, **lock()** and **release()**. These methods are TCMs. Once a field is declared to be of type **locker**, that field can be used to control the execution of TCMs by making calls from the TCMs to *locker.lock()* and *locker.release()*.

If *locker.lock()* is called from multiple TCMs, the first TCM gets the lock and can continue execution. The execution of the other TCMs is blocked. Thus, any resources that are shared between the TCMs are only available to the TCM that gets the lock.

When a TCM calls **release()**, control goes to the next TCM that is waiting on the locker. The order in which the lock is granted is by a FIFO order of client **lock()** requests.

An *e* program uses non-preemptive scheduling, which means thread execution is interrupted only when the executing thread reaches a **wait**, **sync**, TCM call, **release()** **lock()** request, or buffer port operation (**get()** and **put()**). This has two implications:

- a) Locks are not needed unless the code to be executed between the **lock()** and the **release()** contains a **wait**, **sync**, TCM call, or buffer port operation.
- b) Code that is used by multiple threads and is not time-consuming can be put in a regular method, so no locks are needed.

The following restrictions also apply:

- Calling **lock()** again before calling **release()** results in a deadlock. The TCM attempting to acquire the locker stops and waits for the locker to be released. This TCM never executes because it cannot release the locker. Naturally, none of the other TCMs that wait for the locker are executed.
- The release of the locker shall be explicit. If the locking thread ends (either normally or abnormally) without a call to **release()**, the locker is not released. Again, none of the other TCMs that wait for the locked are executed.

Syntax example:

```
lckr.lock();
lckr.release();
```



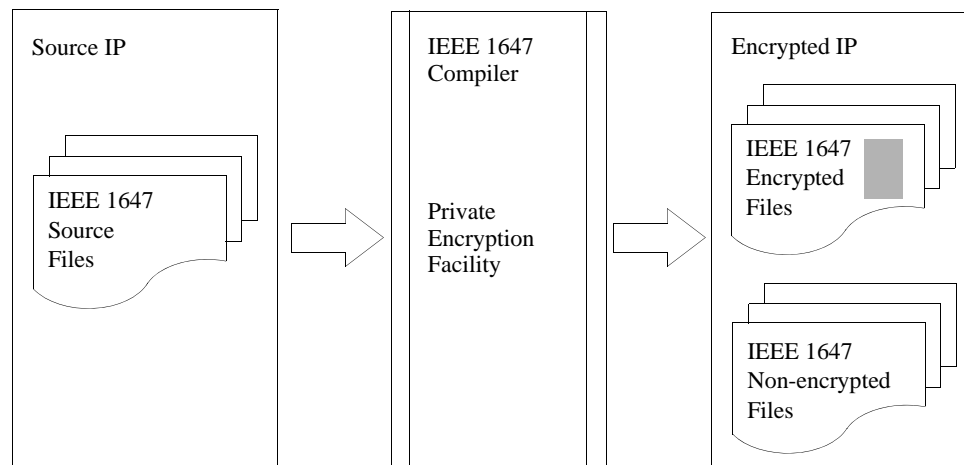
### 33. IP protection

This clause describes Intellectual Property (IP) Protection, which aims to shield valuable information contained in an IP without sacrificing the usability of the IP as a whole. A commonly used method is to transform the clear-text code contained in the IP to a cryptographic representation in order to prevent unauthorized use, review or modification. This method can be used to allow for controlled access to the IP in its encrypted form within an IEEE 1647 compiler.

#### 33.1 Encryption

An IP provider will therefore need to encrypt any sensitive and commercially valuable information contained in the IP before making it available. As the IEEE 1647 LRM does not provide for language constructs that facilitate this process, an IEEE 1647 compiler must provide a facility for encrypting the code contained in the IP.

The IP provider will need to encrypt the IP using the same tool that the end-user will use to decrypt it (see [Figure 21](#)). This is necessary in order to ensure that the integrity of the encrypted information within the IP is not compromised. The IP provider will therefore be required to produce separate versions of the IP for any given IEEE 1647 compiler implementation.



**Figure 21—IP Encryption Flow**

#### 33.2 Decryption

An IEEE 1647 compiler must also be able to decrypt the encrypted IP that was produced with its encryption facility, in order for that IP to be usable (see [Figure 22](#)). The compiler will therefore derive the entire source IP internally during this process, but external access to the originally encrypted information must be restricted for IP protection purposes. As a result, an end-user or higher level tool will still be able to use the IP but will not have access to review or modify any encrypted information without the IP provider's consent.

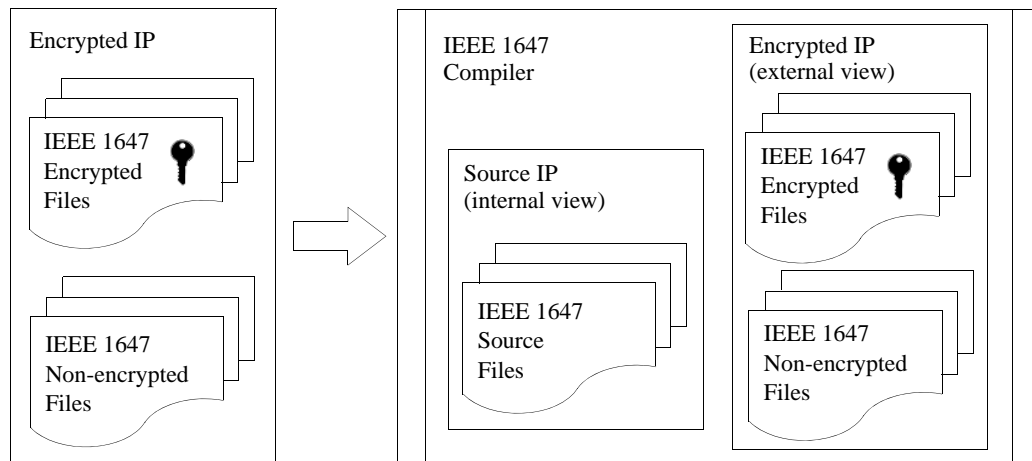


Figure 22—IP Decryption Flow

### 33.3 Reflection API

An IEEE 1647 compiler must export information about whether a module was defined in an encrypted file, by setting the result of the `rf_module.is_encrypted()` method in the Reflection API accordingly. However, since any API can be overridden by the user, such methods should not be used to enforce IP protection features. Furthermore, the compiler must not expose the structure of the encrypted code through general reflection queries, as it may contain sensitive and commercially valuable IP information.

### 33.4 Encryption Targets

While encrypting IEEE 1647 code is critical for IP protection, it can result in significant IP integration and support overheads. Since the end-user does not have access to the entire code in the IP, less debugging information is available. Moreover, modifications and extensions on the IP are harder or even impossible due to the limited information available to the end-user. A set of encryption guidelines and recommendations aimed to minimize this overhead while providing for reasonable IP protection can be found in [Annex F](#).

## Annex A

(informative)

## Bibliography

[B1] Coding examples are available from the IEEE 1647 Working Group Web site: <http://www.ieee1647.org/references.html>.

[B2] Cormen, T., Leiserson, C., Rivest, R., and Stein, C., *Introduction to Algorithms*. MIT Press, 2009.

[B3] IEEE 100, *The Authoritative Dictionary of IEEE Standards Terms*, Seventh Edition. New York: Institute of Electrical and Electronics Engineers, Inc.<sup>13, 14</sup>

[B4] Iman, Sasan, and Joshi, Sunita, *The e Hardware Verification Language*. Springer, 2004.

[B5] Palnitkar, Samir, *Design Verification with e*. Prentice Hall PTR, 2003.

[B6] Piziali, Andrew, *Functional Verification Coverage Measurement and Analysis*. Springer Science+Business Media, LLC, 2008.

[B7] Robinson, David, *Aspect-Oriented Programming with the e Verification Language: A Pragmatic Guide for Testbench Developers*, First Edition. Morgan Kauffman, 2007.

[B8] Wall, L., Christiansen, T., and Orwant, J., *Programming Perl*, Third Edition. O'Reilly, 2000.

<sup>13</sup>IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331, USA (<http://standards.ieee.org/>).

<sup>14</sup>The IEEE standards or products referred to in this annex are trademarks of the Institute of Electrical and Electronics Engineers, Inc.





## Annex B

(normative)

### Source code serialization

This annex addresses two questions:

- a) How is the source code of a program serialized?
- b) How does this affect the semantics of the definition and use of named entities, macros, and preprocessor directives?

The definitions here capture the language rules that correspond to existing practice and code base.

- [B.2](#) gives the first step in answering the preceding question *b*, by considering the simplest case where a module with no **import** statements is loaded.
- [B.3](#) describes the abstract considerations of **import** statement semantics and its effect on definition order and use scopes.
- [B.4](#) addresses question *a* by describing the concrete semantics of **import** statements.

A whole new complication is brought about by the question of visibility scopes of preprocessor rules, which are identifiers declared by **#define** statements. These can be empty (as compilation flags) or have replacement strings (constants). This issue is addressed both in itself, and also because it has direct bearing on the load order, since **import** statements can themselves be inside a **#ifdef** scope. The issue of preprocessor directive visibility rules is addressed in [B.5](#).

#### B.1 The ordering problem in *e*

From a structural viewpoint, a program consists of definitions of named entities (types, fields, methods, etc.) in terms of other entities. Since a program is analyzed in a serial manner, there is the question of how to reference a named entity relative to where it is introduced.

In *e*, the way source code is serialized in its analysis has much farther reaching consequences. This is mainly due to its aspect-oriented (AO) nature, by which the components of a system are described gradually. Since the definition of named entities can be spread between different locations in the source code, the order of analysis of the code not only determines the semantic correctness of the code, but it also affects the behavior of the program. For example, when fields are added to a struct in different extensions, the analysis order of the extensions determines how a struct object is packed or unpacked. An even more typical example is the way the analysis order determines how a method actually runs if it has different extensions in different files.

As a part of the AO modeling paradigm, files are given an important role in the structure of a system. They are treated in *e* as modules; thus, the terms *source file* and *module* are used interchangeably henceforth. This is unlike other languages, in which the separation of code into different files either carries no significance at all (as in C) or else corresponds exactly to the distribution of code between classes (as in Java). So in *e*, the question of serialization becomes the question of how to order source files or modules in the process of analysis.

Another peculiarity of *e*, on which the ordering question has similar bearing, is the ability to extend and modify the syntax language by the program with macros. Here, too, the order can determine whether such modification applies in some context, and thus, effects the correctness of the program or even its behavior.

Two other *e* features are directly connected to order semantics. One is the ability to forward-reference an entity within the module in which it is introduced. This ability is extended to address cyclic dependencies between modules. The other is the *e* preprocessor-like sub-language. The preprocessor rules determine the load order and are actually analyzed by a separate phase according to C-style serialization.

## B.2 Within a single module

Consider a case where a single module with no **import** statements is loaded. Both entities that are declared by the current module and entities that were declared by modules already loaded by previous **load** commands need to be considered here.

### B.2.1 Use scope

The rules on the use of entities within a single module in *e* are more liberal than those of other languages. Obviously, entities declared in modules already loaded can be referred to anywhere in the current module. However, an entity declared in the module itself can be used anywhere within that module. For example:

- A struct can have a field of a struct type declared further down in the source file.
- Constraints can be put on a field that is not declared yet.
- An **enum** item can be presupposed by the implementation of a method and actually be added to the enum type later in the file (by an **extend** statement).
- A **when** subtype can be declared on a field that is actually declared for that struct later in the file.

This permissive policy takes care of the problem of mutually dependent definitions. It releases the language from the need for forward declaration constructs, as is common in other languages. At the same time, it imposes a relaxation algorithm in resolving the references of named entities. References of named entities are resolved in a serial order, which is just the order of the code, but in as many iterations as needed. (A second iteration might not be enough for the resolution of types, since new fields can be introduced under **when** constructs at any depth of nesting and other **when** subtypes might depend on them as well). This iterative resolution process guarantees that whenever there is a resolution, it shall be found.

A further rule is ambiguities shall not arise from forward references. Obviously, when two entities with the same names are declared in the same module, only the second one is reported as an error. More importantly, in the resolution of short name **when** subtype references (such as, “big packet” as opposed to “big size packet”), a declaration of a new field or the addition of a new enum item shall not result in an ambiguity of code in previous lines of the same module.

Unlike named entities, which can be forward-referenced anywhere in the same module, macros apply to code in the same module from the point of definition onwards. Macros cannot be forward-referenced, since they are purely syntactic rules; there is no entity they introduce that can be referenced.

### B.2.2 Definition order

Some named entities in *e* are extensible in the sense they can be declared and defined initially at one place in the program’s source code, and then their definition can be extended in other places. Extensible entities in *e* are structs, methods, events, and **enum** types. The initial definition, and each of the extensions of such entities, is given by some single linguistic construct (e.g., in the case of structs, the **struct** and **extend** statements). The part of the definition given by a single construct is called a *layer* of the definition. More than one layer of the definition of the same entity can be located in the same module and different layers can be located in different modules.

The order of layers in the definition of an entity counts for more than just resolution of reference, so here the straightforward rule applies. A new layer that is added in the module currently loading to an entity, which was declared previously, comes after any layers in the modules already loaded. The order of layers of the same entity in the currently loading module depends on the order of the constructs' appearance in the source file. Here, as opposed to the use scope of names, the declaration shall appear before any extension, even within the same module. For example, a struct type can be forward-referenced in the declaration of a variable, but the struct cannot be extended or inherited before the struct itself is declared.

Following are a few examples of the significance of order within definition layers:

- When a method is extended twice in the same module with the **is also** modifier, the extension that appears last in the source file shall run last when the method is called.
- If a method declared by previous modules is extended with the **is first** modifier, this code shall run before any other, including any layers of that method defined by a super-type of that struct.
- When **enum** items are added to an enum type by two different statements (a **type** and an **extend** statement) the values of the last items added take the subsequent integer values (unless they are explicitly assigned).

### B.3 Importing and dependency

Definitions in one module make use not only of entities defined within it or in *e*'s core library, but also of entities declared in other modules. The **import** statement declares that one module relies on the declarations of another module. This statement guarantees the imported module is loaded before (or along with) the importing module. Thus, an **import** statement declares *direct dependency* between two modules, and *dependency*, in general, is the transitive closure of the direct dependency relation.

The dependency relation is not necessarily asymmetric. Sometimes the definitions in one module presuppose declaration of another module and vice versa. In such cases, whichever way the definitions in these modules are serialized, a forward reference occurs in the use of some named entity across the “module boundary.” Therefore, in cyclic dependencies, the code is actually treated as if it were all located in a single module in the sense previously described, namely being a single use scope of entities. In other words, entities declared in any of the mutually dependent modules can be used anywhere within these source files. Therefore, code in modules that depend on each other need to be serialized so no other module comes in between them. So, cyclic dependencies affect load order.

These considerations call for the introduction of a generalized concept—a dependency unit. *Dependency units* are single modules or sets of mutually dependent modules (identified by **import** statements). One dependency unit depends on another when a module of that dependency unit imports a module of the other dependency unit. Unlike dependency between modules, the dependency relation between dependency units is asymmetric and can be sorted topologically. Rephrasing the semantics of an **import** statement: it guarantees the imported module is loaded previously or in the same dependency unit as the importing module.

There are three requirements on the load order of modules, as follows:

- a) If module *a* depends on module *b*, but *b* does not depend on *a*, module *b* loads before module *a*.
- b) Modules in a single dependency unit load in consecutive order. More precisely, when modules *a* and *b* depend on each other and module *c* loads between them, then modules *a* and *c* also depend on each other.
- c) Whenever possible, the order of **import** statements in the source code needs to be taken into consideration. Module *a* should load before module *b* if both are imported by module *c* and the **import** statement of *a* appears in the source code before the import statement of *b*. This is true only

when module  $c$  is the first module that imports module  $b$  and circular dependencies do not require otherwise.

These requirements can be taken as the user-view definition of the load order determined by **import** statements. Any ordering of modules that satisfies requirements [a\)](#) and [b\)](#) is, in principle, a legitimate implementation of the semantics of the **import** statement from the user's viewpoint. The extent to which consideration [c\)](#) plays a role in the definition, however, remains open.

In particular, rules [a\)](#) and [b\)](#) do not fully determine the load order of modules within a dependency unit, on the one hand, or the order between unrelated dependency units, on the other. Consideration [c\)](#) might reduce the indeterminacy, but still leave room for a different ordering.

Ideally, this is not something the user needs to know. The correctness and behavior of a program that is well designed in terms of aspect orientation should not be affected by the different sorting of its dependency unit or ordering of modules within a dependency unit. However, to guarantee used entities are in scope, explicitly declare any dependencies by using **import** statements.

## B.4 Concrete load order

The module that is explicitly mentioned in the **load** command (or equivalently a single compilation) is called the *root module* of that load. The set of modules that are loaded by a single **load** command is called a *load cluster*. These are all the modules upon which the **root** module depends, except those that are already loaded. The order for loading modules during the execution of the **load** command is uniquely determined by the code in the files of the load cluster. Modules that are already loaded are ignored in this process, since their source files might not even be available.

Consider a directed graph, where the modules of some load cluster are nodes and the **import** statements correspond to edges from the importing module to the imported one. This is called the *import graph* for a given root module. Here, the description and correctness proof of the strongly connected component (SCC) algorithm from *Introduction to Algorithms* [\[B2\]](#) is presupposed and the same terminology is being employed.

- a) *Discovery time* and *finish time* are added to each node in the import graph using a simple DFS. The algorithm starts from the **root** module, where the exploration order of modules adjacent to a given module corresponds to the order of the **import** statements in the source file [consideration [c\)](#) in [B.3](#)].
- b) The SCCs of the graph are found, which correspond to dependency units. Collapsing all nodes in a SCC to a single node and keeping all edges between SCCs leaves a directed acyclic graph (DAG), which can be called the SCC-DAG.

Load ordering according to any topological sort on the SCC-DAG yields requirement [a\)](#) (in [B.3](#)), and any consequent order within each SCC is just requirement [b\)](#) (in [B.3](#)).

- c) The load order of the whole load cluster is determined by using a DFS on the SCC-DAG. This DFS starts from the SCC of the **root** module and explores new SCCs in the following order: It starts from edges that originate from nodes with the greatest *finish time* and proceeds to edges originating from nodes in a descending *finish time* order. The edges originating from the same node are traversed in an order corresponding again to the order of the **import** statements in the module. After allocating load time to all dependent SCCs, the order of the modules inside the current one can then be determined, which is by descending *finish time*.

Now, the following definitions also serve to describe the algorithm in pseudo-code:

$IG$  is an import graph.

$V[IG]$  is the set of all modules in the import graph—the load cluster.

$root[IG]$  is the root module of the load cluster.

$Adj[u]$  is the set of all modules imported by module  $u$ .

$d[u]$  and  $f[u]$  are respectively the discovery time and finish time of vertex  $u$  calculated by the DFS on the import graph.

$SCC[u]$  is the strongly connected component to which vertex  $u$  belongs, as is calculated by the SCC algorithm.

Plus, an array *color* (with the usual meaning) is used for each vertex and a global variable *time*:

Calculate-Load-Order(IG)

```
for each  $u \in V[IG]$  do
  color[u]  $\leftarrow$  WHITE
time  $\leftarrow$  0
Visit-SCC(SCC[root[IG]])
```

Visit-SCC(c)

```
color[c]  $\leftarrow$  GRAY
for each  $u \in c$  in descending  $f[u]$  order do
  for each  $v \in Adj[u]$  (in the import statement order) do
    if color[SCC[u]] = WHITE
      Visit-SCC(SCC[u])
for each  $u \in c$  in descending  $f[u]$  order do
  load-time[u]  $\leftarrow$  time  $\leftarrow$  time+1
color[c]  $\leftarrow$  BLACK
```

The resulting load order is given in the *load-time* array, where each module has a unique index in the load process. Thus, the order can be determined: module  $u$  loads after module  $v$  **iff**  $load-time[u] < load-time[v]$ .

NOTE 1—If there are no circular dependencies, each SCC consists of just one module. The load order in this case is simply the ascending order of the finish time, since the algorithm runs just like a simple DFS.

NOTE 2—A circular dependency breaks this intuitive ordering, since the nodes from one SCC to another are not explored in an order corresponding to that of the **import** statements of the corresponding source file. Specifically, adding an **import** statement at some file in a big design might affect the load order globally and not just for modules dependent on it.

## B.5 Visibility scope of preprocessor directives

This subclause describes the issue of preprocessor directive visibility rules.

### B.5.1 Overview

The **#define** statement in *e*, along with **#ifdef**, **#ifndef**, **#else**, and **#undef**, is intended to be used in a way similar to that of C preprocessor. This makes the visibility scope of **#define** rules very different from “real” *e* entities—both named entities and macros.

So far, only the order that figures in the use of named entities and macros has been discussed. This can be called the *load order* to distinguish it from the different conception of ordering that is involved in determining the visibility scope of **#define** statements. This order is given by treating the **import** statements just like a C preprocessor treats the **#include** directive. A **#define** statement that appears before an **import** statement is visible by, or applies to, all the code in the imported file (unless that module was loaded previously), although in terms of load order the declaration of the **#defined** name actually comes after it. This order is called the *include order*.

In general, the same source files are analyzed by two separate phases. The preprocessing phase is responsible for the discovery of the dependency relation. It figures out the load cluster and the order within it. It also executes the preprocessor directives, but does so according to the *include order*, as the *load order* is not yet known.

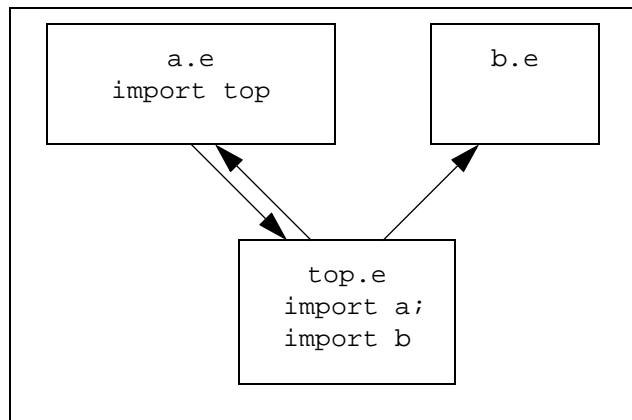
The second phase is the actual parsing of the full *e* code and its analysis. This phase imposes a serialization on the source code that can differ from the first phase. This discrepancy between the scopes of applicability of different statements in *e* has some unintuitive consequences that are demonstrated in the following subclauses.

## B.5.2 Cases where order differs

There are two patterns where the include order is different from the load order, and so, the scope of application of the **#define** statements is reversed.

### B.5.2.1 Case 1

The simplest case occurs where the cyclic import rule [requirement b) in [B.3](#)] overrides the **import** statement order in the source file [requirement c) in [B.3](#)], as shown in [Figure B.1](#).

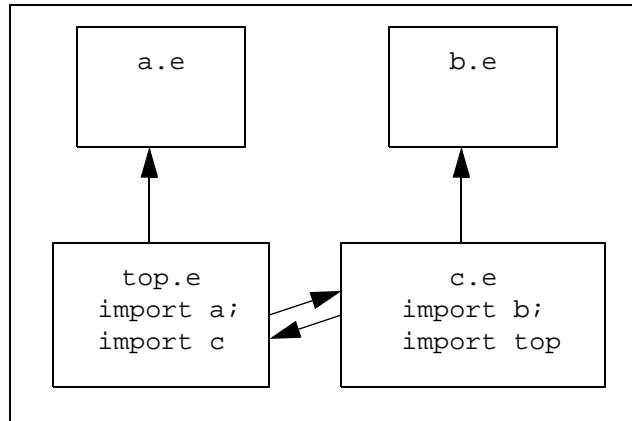


**Figure B.1—Overriding the import statement**

Here module *b* loads before module *a*, even though *a* is discovered first in the DFS. Definitions in *b* are available in *a* if they obey load order, but not if they obey include order, and vice versa.

### B.5.2.2 Case 2

This case (see [Figure B.2](#)) shows the effect of *e*'s concrete ordering (see [B.4](#)), which is not inherent in the abstract requirements.



**Figure B.2—Ordering reversals without cyclic dependencies**

Here, module *a* was discovered before module *b*, but loads after it. In this case, too, definitions in *b* are available in *a* if they obey load order, but not if they obey include order. What is interesting about this example is the fact that the two modules whose ordering reverses have no **import** statements (or at least none relating to the cycle) and so they have no trace of cyclic dependency.

### B.5.3 Examples

The following examples demonstrate the surprising consequence of [B.3](#), [B.4](#), and [B.5.1](#).

#### *Example 1*

This uses Case 1 (see [B.5.2.1](#)) to show how the order of definition and use of the preprocessor versus proper *e* is reversed.

##### **top.e**

```
<
import a;
import b
>
```

##### **a.e**

```
<
import top;
#define A_SCANNED;
type t_a : t_b // 't_a' definition presupposes 't_b' definition
>
```

##### **b.e**

```
<
#ifdef A_SCANNED {
type t_b : int // 't_b' definition presupposes the C-like
               // define 'A_SCANNED'
}
>
```

In this test case, it might seem that whichever way the code in modules *a* and *b* are ordered, module *a* will fail to load. But it does load, since the **#define** directive of `A_SCANNED` precedes the **#ifdef** statement according to *include order*, even though the type declaration of `t_b` precedes its use in the declaration of `t_a` according to the *load order*. If module *a* did not import module *top*, the load order falls back to the **#include** order and the code fails to load.

### Example 2

This uses Case 2 (see [B.5.2.2](#)) to show how the definition of syntactic rules with *e* macros applies according to the load order and how this can be made to stand in reverse order compared to a preprocessor C-like define statement.

#### **a.e**

```
<
#define A_SCANNED;
sys_add_field foo
>
```

#### **b.e**

```
<
#ifdef A_SCANNED {
  define <sys_add_field'statement> "sys_add_field <name>" as {
    extend sys {
      <name> : int
    }
  }
}
>
```

#### **c.e**

```
<
import b;
import top
>
```

#### **top.e**

```
<
import a;
import c
>
```

This also loads perfectly well. But, if the cyclic dependency between modules *top* and *c* is removed (by commenting out the second line in *c.e*), the code fails to load because the load order between modules *a* and *b* reverses.



## Annex C

(informative)

### Comparison of when and like inheritance

There are two ways to implement object-oriented (OO) inheritance in *e*:

- *Like inheritance* is the classical, single inheritance familiar to users of all OO languages.
- *When inheritance* is a concept introduced by *e*. It is less familiar initially, but lends itself more easily to the kind of modeling done in *e*.

This annex discusses the pros and cons of both these types of inheritance and recommends when to use each of them.

#### C.1 Summary of when versus like

In general, “when” inheritance should be used for modeling all DUT-related data structures. It is superior from a knowledge representation point of view and from an extensibility point of view. When inheritance can

- explicitly reference a field that determines the **when** subtype.
- create multiple, orthogonal subtypes.
- use random generation to generate lists of objects with varying subtypes.
- easily extend the struct later.

Although like inheritance has more restrictions than when inheritance, it is recommended in some special cases because of the following:

- a) Like inheritance is somewhat more efficient than when inheritance.
- b) Generation of objects that use like inheritance can also be more efficient.

##### C.1.1 A simple example of when inheritance

A **when** subtype of a generic struct can be created using any field in the struct that is a Boolean or enumerated type. This field, which determines the **when** subtype of a particular struct instance, is called the *when determinant*.

*Example 1*

In the following example, the when determinant is `legal`.

```
struct packet {
    legal : bool;
    when legal packet {
        pkt_msg() is {
            out("good packet")
        }
    }
}
```

NOTE—The following syntax is used in this document because it looks closer to the “like” version:

```
extend legal packet {...}
```

This syntax is exactly equivalent to the **when** construct:

```
extend packet {when legal packet {...}}
```

### Example 2

The following example shows a generic packet struct with three fields—protocol, size, and data—and an abstract method `show()`. In this example, the `protocol` field is the determinant of the when version of the packet, i.e., this field determines whether the packet instance has a subtype of `IEEE`, `Ethernet`, or `foreign`. In this example, the `Ethernet packet` subtype is extended by adding a field and extending the `show()` method.

```
type packet_protocol : [Ethernet, IEEE, foreign];

struct packet {
    protocol    : packet_protocol;
    size        : int [0..1k];
    data[size]  : list of byte;
    show() is undefined // To be defined by children
};

extend Ethernet packet {
    e_field : int;
    show() is {
        out("I am an Ethernet packet")
    }
}
```

### Example 3

Of course, it is possible for a struct to have more than one when determinant. In the following example, the `Ethernet packet` subtype is extended with a field of a new enumerated type, `Ethernet_op`.

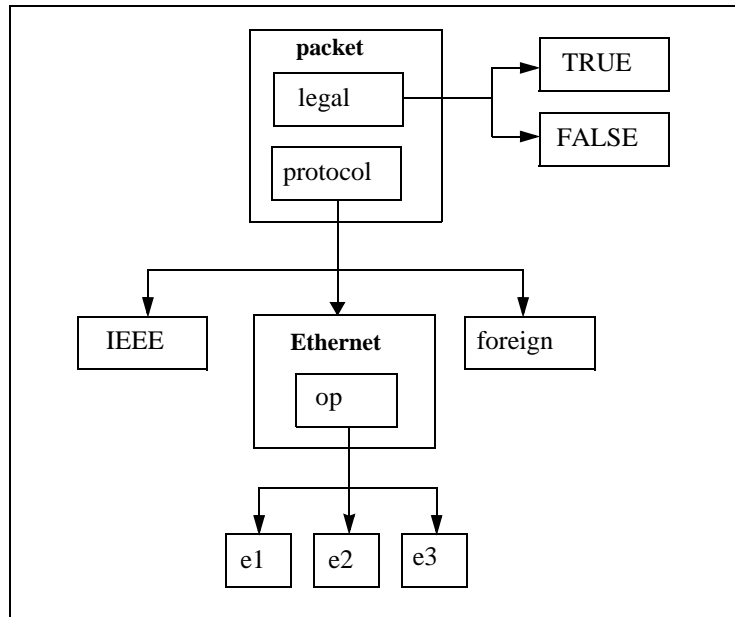
```
type Ethernet_op : [e1, e2, e3];

extend Ethernet packet {
    op : Ethernet_op
};

extend e1 Ethernet packet {
    e1_foo : int;
    show() is {
        out("I am an e1 Ethernet packet")
    }
}
```

Because it is possible for a struct to have more than one when determinant, the inheritance tree for a struct using when inheritance consists of any number of orthogonal trees, each rooted at a separate enumerated or Boolean field in the struct. [Figure C.1](#) shows a when inheritance tree consisting of three orthogonal trees rooted in the `legal`, `protocol`, and `op` fields.

NOTE—The **when** subtypes that have not been explicitly defined, such as `IEEE packet`, exist implicitly.



**Figure C.1—When inheritance tree for packet struct subtypes**

### C.1.2 A simple example of like inheritance

A like child of a generic struct can be created by using the **like** construct. In the following example, a child `Ethernet_packet` is created from the generic struct `packet` and is extended by adding a field and extending the `show()` method.

*Example*

```

struct packet {
    size      : int [0..1k];
    data[size] : list of byte;
    show() is undefined // To be defined by children
};

struct Ethernet_packet like packet {
    e_field : int;
    show() is {
        out("I am an Ethernet packet")
    }
}
  
```

In the same way, an `IEEE_packet` can be created from `packet` using **like**:

```

struct IEEE_packet like packet {
    i_field : int;
    show() is {
        out("I am an IEEE packet")
    }
}
  
```

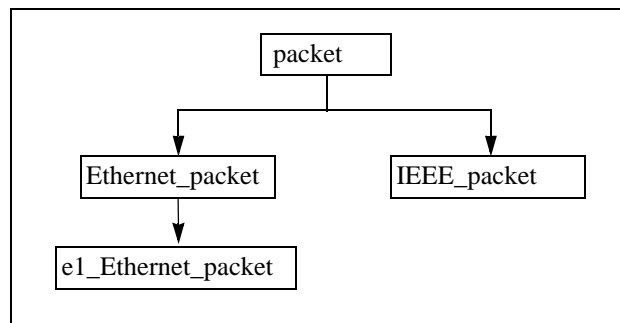
Or an `e1_Ethernet_packet` can be created from `Ethernet_packet` using like inheritance.

```

struct e1_Ethernet_packet like Ethernet_packet {
    e1_foo : int;
    show() is {
        out("I am an e1 Ethernet packet")
    }
}

```

In contrast to the when inheritance tree, the like inheritance tree for the packet type is a single tree where each subtype needs to be defined explicitly, as shown in [Figure C.2](#). This difference between the like and when inheritance trees is the essential difference between like and when inheritance.



**Figure C.2—Like inheritance tree for packet struct subtypes**

## C.2 Advantages of using when inheritance for modeling

While the like version and the when version look similar, and the “like” version might seem more natural to those familiar with other OO languages, the “when” version is much better for the kind of modeling typically done in *e*. There are several reasons for this, as follows:

- Determinant fields can be explicitly referenced.
- Multiple orthogonal subtypes can be used.
- Lists of objects with varying subtypes can be used.
- The struct can be extended later.
- A new type can be created by simple extension.

### C.2.1 Determinant fields can be explicitly referenced

In the when version, the determinant of the when is an explicit field. In the like version, there is no explicit field that determines whether a packet instance is an Ethernet packet, an IEEE packet, or a foreign packet. The explicit determinant fields provide several advantages.

- Explicit determinant fields are more intuitive.  
Fields are more tangible than types and correspond better to the way hardware engineers perceive architectures. Having a field whose value determines what fields exist under it is familiar to engineers. (It is similar to C unions, for example.)
- Attributes of determinants that are physical fields can be used.

If the determinant is a physical field, it might be desirable to specify its size in bits, the mapping of enumerated items to values, where it is in the order of fields, and so on. These things are done very naturally with when inheritance, because the determinant is just another field. For example:

```
%protocol : packet_protocol (bits:2)
```

- With like inheritance, a field can be defined similarly to the when determinant, but it also needs to be tied into the type with code similar to the following:

```
var pkt : packet;

case protocol {
  Ethernet {var epkt : Ethernet packet; gen epkt; pkt = epkt};
  IEEE     {var ipkt : IEEE packet;   gen ipkt; pkt = ipkt}
}
```

Plus, there is an added inconvenience of having to generate or calculate protocol separately from the rest of the packet.

- The when determinant can be constrained.

Using when inheritance, it is very natural to write constraints like these in a test:

```
keep protocol in [Ethernet, IEEE];
keep protocol != IEEE;
keep soft protocol == select {
  20 : IEEE;
  80 : foreign
};
keep packets.is_all_iterations(.protocol, ...)
```

Constraining the value of fields in various ways is a main feature of generation. Doing the same with like inheritance is more complicated. For example, the first constraint above could be stated something like this:

```
keep me is an Ethernet_packet or me is an IEEE_packet
// This pseudocode is not a legal constraint specification
```

However, constraints such as this can become quite complex in like inheritance. Furthermore, there is no way to write the last two constraints.

## C.2.2 Multiple orthogonal subtypes can be used

Suppose each packet (of any protocol) can be either a normal (data) packet, an ack packet, or a nack packet, except that foreign packets are always normal.

*Example*

```
type packet_kind: [normal, ack, nack];

extend packet {
  kind : packet_kind;
  keep protocol == foreign => kind == normal
};

extend normal packet {
  nl : int
}
```

How can this be done in like inheritance (disregard for now the issue of extending the packet struct later)?

- a) Assuming the requirement previously stated is known in advance and should be modeled using like inheritance in the best possible way:

```
struct normal_Ethernet_packet like Ethernet_packet {
    n1 : int
};

struct ack_Ethernet_packet like Ethernet_packet { ... };
struct nack_Ethernet_packet like Ethernet_packet { ... };
struct normal_IEEE_packet like IEEE_packet { ... }
```

This requires eight declarations.

- b) Then, the Ethernet\_op possibilities need to be taken into account:

```
struct ack_el_Ethernet_packet like el_Ethernet_packet { ... }
```

This works, but requires  $((N_1 * N_2 * \dots * N_d) - \text{IMP})$  declarations, where  $d$  is the number of orthogonal dimensions,  $N_i$  is the number of possibilities in dimension  $i$ , and IMP is the number of impossible cases.

- c) Another issue is how to represent the impossible cases.

Multiple inheritance would solve some of these problems, but would introduce new complications.

With when inheritance, all the possible combinations exist implicitly, but they do not have to be enumerated. It is only when something needs to be specified about a particular combination that it is enumerated, as in the following examples:

```
extend normal IEEE packet { ni_field : int }; // Adds a field
extend ack el Ethernet packet { keep size == 0 } // Adds a constraint
```

All in all, the when version is more natural from a knowledge representation point of view, because of the following:

- It is immediately clear from the description what goes with what.
- Types only need to be specified if there is something to say about them.

### C.2.3 Lists of objects with varying subtypes can be used

The job of the generator is to create (in this example, packet instances). By default, all possible packets are generated. In both versions, a list of packets is created, e.g.,

```
extend sys { packets : list of packet }
```

However, the generator should only generate fully instantiated packets. In the when version, that happens automatically—there is no other way.

With like inheritance, when a parent struct is generated, only that parent struct is created; none of the **like** children are created. For example, the following **gen** action always creates a generic packet, never an Ethernet packet or an IEEE packet:

```
pkt : packet;
gen pkt
```

Thus, in practice, only fields whose type is a leaf in the like inheritance tree should be generated, e.g.,

```
p : el_Ethernet_packet;
```

gen p

### C.2.4 The struct can be extended later

There are some restrictions on extending structs that have like children, however; see [6.3](#) for more details.

### C.2.5 A new type can be created by simple extension

The following example extends the `packet_protocol` type and adds new members to the `packet` subtype:

```
extend packet_protocol : [brand_new];

extend brand_new packet {
  ...new struct members...
}
```

The old environment is automatically able to generate the `brand_new` packets. With like inheritance, all instances of the procedural generation code need to be identified first and then a new case needs to be added to the case statement.

## C.3 Advantages of using like inheritance

Like inheritance is a shorthand notation for a subset of when inheritance. It is restricted, but more efficient. Like inheritance often has better performance than when inheritance for the following reasons:

- Method calling is faster for like inheritance.
- When generation is slower than like generation. This can be important if a large part of the total run-time is attributable to generation.
- When inheritance uses more memory, because all of the fields of all of the when subtypes consume space all the time.

NOTE—If this becomes a problem in a particular design, there is a workaround. Rather than having many separate fields under the **when**, put all the fields into a separate struct and put a single field for that struct under the **when**. For example, the following coding style could use a lot of memory if there are many fields declared under the Ethernet packet subtype:

```
type packet_protocol : [Ethernet, IEEE, foreign];

struct packet {
  protocol : packet_protocol;
  when Ethernet packet {
    e_field0 : int;
    e_field1 : int;
    e_field2 : int;
    e_field3 : int;
    // ...
  }
}
```

A more efficient coding style follows, where a single field is declared under the `Ethernet_packet` subtype.

```
type packet_protocol : [Ethernet, IEEE, foreign];
```



```

struct Ethernet_packet {
    e_field0 : int;
    e_field1 : int;
    e_field2 : int;
    e_field3 : int;
    // ...
};

struct packet {
    protocol : packet_protocol;
    when Ethernet packet {
        e_packet : Ethernet_packet
    }
}

```

## C.4 When to use like inheritance

Like inheritance should be used for modeling only when the performance win is big enough to offset the restrictions, e.g.,

- For objects that use a lot of memory, such as a register file, where the number of distinct registers is very large, and for each such register a field of the register type needs to be generated, such as, `pc: pc_reg, psr: psr_reg`, and so on.
- For objects that do not require randomization, such as a scoreboard or a memory.

Like inheritance should also be used for non-modeling, programming-like activities, such as implementing a generic package for a queue.



## Annex D

(normative)

### Name spaces

This annex defines an extension to the encapsulation scheme—the scoping of names. The naming problem in *e* is addressed by using packages as *name spaces* for types. This overloading of the notion of a package is natural and common in other languages.

#### D.1 The naming problem in *e*

Currently, all types and structs in *e* share one global name space, in which every name needs to be unique. This problem is more acute in *e* than in other programming languages, because there are no separate binary libraries or object files. As verification projects grow in size and depend more on components developed externally, the risk of name collisions increases. Such name clashes can be hard to work around, especially when they occur during integration of code from different parties.

Fields, methods, and events shall have a unique name in the context of their structs, including those added in distant extensions of the struct. Similarly, **enum** items shall have a unique identifier within a single **enum** type, even when they are added by later extensions. Name collisions might occur also for these entities, although they are much less likely.

In *e*, this problem is addressed by using a naming convention that assigns a globally unique name to every type and a unique name to every struct member or enum item in extensions outside of the declaring package. The unique name consists of the simple name of the entity prefixed by the package name, e.g., the unit that represents a monitor in the `vr_xbus` package is named `vr_xbus_monitor_u`.

The problem with this convention is most names become very long; so the code can become cumbersome and unreadable. The readability of the output can also be impacted. Another problem is the convention is not enforced by an *e* compiler; thus, it becomes the developer's responsibility not to pollute the global name space with unqualified names.

#### D.2 Resolution overview

This material previews the resolution for using name spaces in *e*.

##### D.2.1 Packages as name spaces

Packages, being the major encapsulation vehicle in *e*, are the natural candidates for serving as name spaces. As in the *e* naming convention, package names serve to qualify type names declared within their context. But, unlike the *e* naming convention, this solution involves full linguistic support for name qualification. The support consists of syntactic differences between qualified and unqualified type names, and semantic rules for resolving the reference of unqualified names.

##### D.2.2 Name spaces for other named entities

According to *e*, fields, methods, events, and **enum** items shall qualify their names with the package name only when declared in extensions outside the package where their context type is declared. These cases are

relatively rare, so keeping to the convention does not affect the readability of the code significantly. Furthermore, even when the convention is not strictly kept, the probability of a name collision is low. On the other hand, having packages serve as name spaces for struct members or **enum** items is unintuitive and hard to define. Therefore, the name space scheme described here is restricted to types only.

### D.2.3 Name resolution

Types can be referenced by their given name or a qualified name. Referencing a type by a qualified name succeeds given a type by that name exists in the right package (in code that is already loaded). As is required by the compatibility constraint, unqualified references succeed also if only one type by that name exists in some package. In the case of unqualified references where more than one type by the same name exists in different packages, the compiler tries to resolve the reference according to set priorities. A type declared within the context package has the highest priority. Public types declared in packages used by the context package are next in priority. Only then follow all the rest, i.e., types in packages not used by the context package. See also [D.3](#).

### D.2.4 The use relation

**import** statements are used to declare dependencies between different parts of a design; they determine if a package uses another package. A package uses another package when one of its modules *directly* imports one of the other package's modules. An important consequence of this definition is the use relation is defined in terms of packages, which puts more weight on the package as a whole. This goes together well with the idea of using packages as encapsulation units and name spaces as encapsulation mechanisms. The downside of this is every inter-package **import** statement affects the name resolutions of the whole package. This may force unnecessary qualification of names in the modules. For more details, see [D.4](#).

### D.2.5 Reserved type names

One set of types in *e* is considered essential or “core,” e.g., **int**, **string**, and **sys**. No reasonable *e* program would assign the name of any of those types to a user-defined type. Core types are declared under a special package called **e\_core**. Within the name space model, **e\_core** type names are reserved. Unlike other type names, they cannot be used in another package. For more information, see [D.3.3](#).

## D.3 Qualified and unqualified names

Types (scalars or structs) declared inside a package belong to the name space of that package. Type names shall be unique in the context of the package, but types in different packages can have the same name. Fields, methods, and events shall have unique names within their context struct even if they are declared in extensions in different packages. The same goes for **enum** items in the context of the **enum** type. Many other built-in derivatives of explicitly defined types, such as lists and size-modified scalars, are available to a program. These are not associated directly with a package, but rather through their explicit base type.

### D.3.1 Name rules

- An explicitly declared type can be referenced using an *unqualified name* (an *e* identifier) or a *qualified name* in the form:

```
id :: id
```

The second identifier is the type's given name, and the first is the package in which it was declared. The double colon (`::`) is called the *scope operator*.

NOTE—The scope operator does not relate directly to built-in derivatives of a type. Rather, it relates to the explicitly declared type that serves as the base for the derivatives. For example, the qualified name of `list of packet` would be `list of vr_xbus::packet` and not `vr_xbus::list of packet`. Similarly, in the case of **when** qualifiers, `big corrupt packet` would be `big corrupt vr_xbus::packet`.

- Only an unqualified name can be used in the declaration of a new type (a **type**, **struct**, or **unit** statement).
- In any other construct where type names matter, both qualified and unqualified names are syntactically legal.
- Whether the module is associated explicitly with a package (by a **package** statement) or implicitly with package **main** (if there is no **package** statement) is irrelevant to naming, i.e., types declared in modules that do not explicitly associate themselves with a specific package are part of the name space of package **main**.
- The names of types declared in the package **e\_core** are reserved and cannot be used in the declaration of new types anywhere.

### D.3.2 Type reference resolution

- A qualified name always fully determines the reference. If there is no type by the given name in the given package, a *type not found* error is issued.
- The reference of unqualified names is determined on the basis of the following three priorities (listed in order of priority):
  - 1) A type declared within the context package
  - 2) A public type declared in a package that is used by the context package (see [D.4](#))
  - 3) A public type declared in packages outside of the current context (priorities 1 and 2)
- If more than one public type with the same name and priority is found during the search process (this is only possible for priorities 2 and 3), then an ambiguity error is issued. In this case, the user needs to resolve the ambiguity by qualifying the type name.

### D.3.3 e\_core types

The set of types declared in the built-in package **e\_core** is privileged. Unlike all other types, the names of the **e\_core** types remain unique, even if they have been added by user extension. When a user tries to define a type using the name of an **e\_core** type, an error is reported. However, these names are not reserved keywords; they can be used as identifiers in any other context (variable names, method names, and so on).

The **e\_core** types are: **int**, **uint**, **byte**, **bit**, **bool**, **string**, **sys**, **global**, **base\_struct**, **any\_struct**, **any\_unit**, and **event\_port**.

## D.4 Use relation

An **import** statement declares the dependency of one module on another. These dependencies determine the load order of modules in a single **load** command or compilation (see [Annex B](#)). As part of the name space scheme, a use relation between packages is defined in terms of *module dependency*. A package uses another package if any modules belonging to the former directly import modules belonging to the latter. As explained in [D.3.2](#), packages used by the current module's package are second in the search list for type resolution, after the context package itself, but before the rest of the packages.

### D.4.1 Load clusters

The definition of the use relation requires some amplification. Packages are not necessarily loaded all at once; new modules of already known packages can be loaded at any stage. Hence, when determining which packages are used by a module being loaded at a given stage, some modules of a specific package may not be taken into account.

On the other hand, it is not enough to take into account only modules that have already been loaded. A typical case would be when the top file of package *A* imports the top file of package *B* and then imports the rest of package *A*'s files. Package *A*'s files are actually loaded before the top file is, but would naturally presuppose package *B* as being used already in their context. For this reason the concept of a load cluster is needed.

A *load cluster* is the set of modules that are loaded by a single **load** command (or a single compilation). All modules that together form a load cluster depend on a common root in the dependency graph — the module explicitly mentioned in the **load** command. Consequently, package *A* uses package *B* at a given stage in the load process if there is a module *m* of package *A* that is either already loaded or is part of the current load cluster and *m* directly imports some module of package *B*.

NOTE 1—A module that is imported by a module in the current load cluster might be already loaded by previous **load** commands. Whether or not it is already loaded does not affect the use relation.

NOTE 2—If more than one module is named by a **load** command or a single compilation (e.g., `load a.e, b.e`), the two modules with all of their dependencies are treated as a single load cluster.

### D.4.2 Examples

These examples demonstrate various use relationships.

#### D.4.2.1 Example 1

This example illustrates a simple use relation, as shown in [Figure D.1](#).

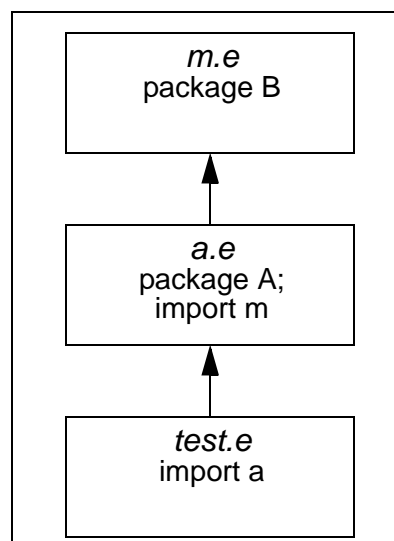


Figure D.1—Simple use relation

Since module *test.e* is the one explicitly loaded, and thus is the root of the load cluster, package *B* uses package *A* and package *main* (the package of module *test.e*) uses package *B*. Type references would be resolved accordingly.

#### D.4.2.2 Example 2

This example (see [Figure D.2](#)) shows how the use relation takes into account the whole load cluster.

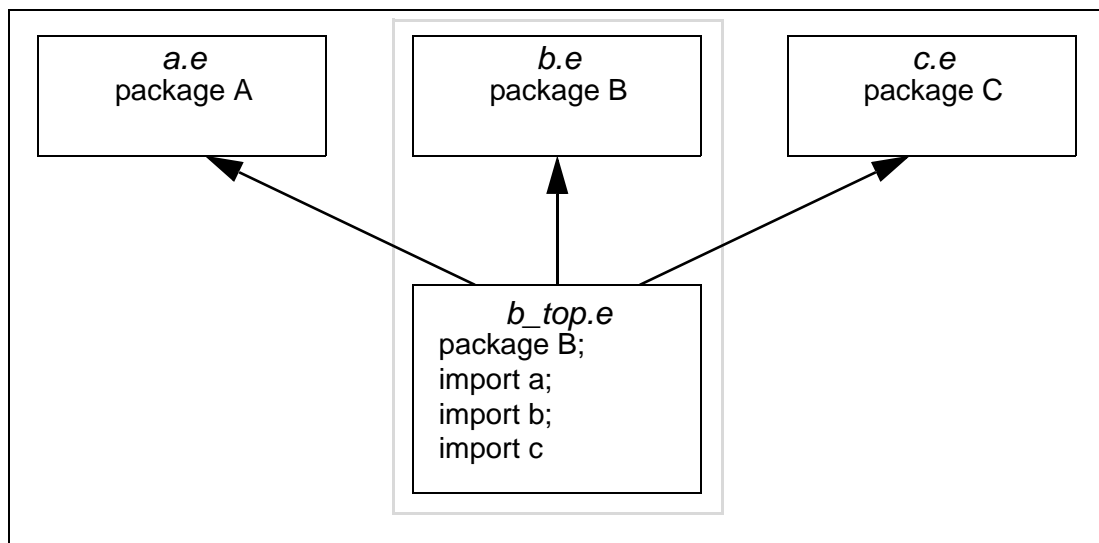


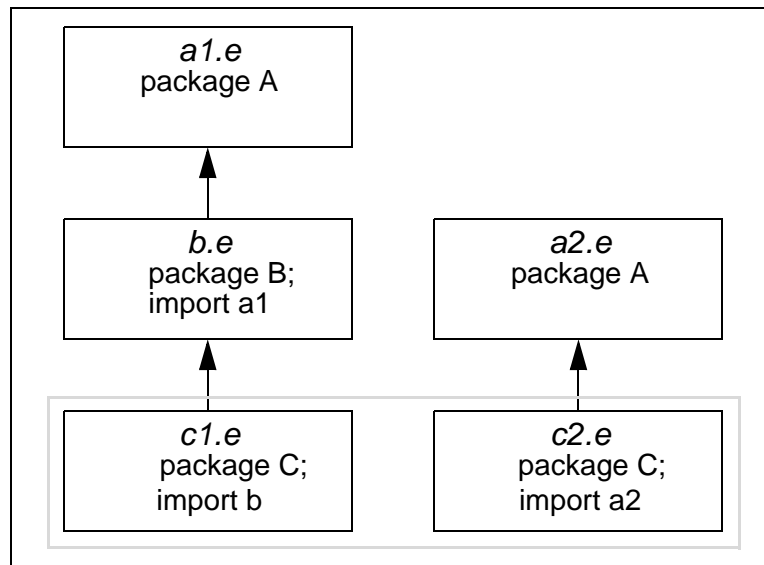
Figure D.2—Use relation for a load cluster

Since module *b\_top.e* is the root of the load cluster, package *B* uses both package *A* and package *C*. Thus, in resolving type names in module *b.e*, both *A* and *C* are name spaces with priority over packages outside of the current context, even though this module is loaded before module *b\_top.e* (but only *after* modules *a.e* and *c.e* are imported).

NOTE — Types (like any other named entities) declared in *c.e* could not be used in *b.e*, because they are still not loaded. Of the named entities declared within package *C*, only those in modules that are previously loaded can be referenced at all, that is, with or without qualification.

#### D.4.2.3 Example 3

This example (see [Figure D.3](#)) shows how previous **load** commands affect the use relation in the current load cluster, but not vice versa.



**Figure D.3—Use relation dependencies**

Module *c1.e* is explicitly loaded by a **load** command, and module *c2.e* is loaded by a subsequent **load** command. In this case, when *c1.e* is loaded, package *C* is only using package *B*. Names in package *B* shadow names in package *A* in the context of *c1.e*. When module *c2.e* is loaded, both package *A* and *B* are used by package *C*; therefore, a type name declared in both *A* and *B* with the same name would be ambiguous in the context of *c2.e*.

## D.5 Built-in APIs

This subclause defines how name spaces can impact (using) APIs.

### D.5.1 Reflection and name spaces

The reflection API deals with type names in two ways: getting a type with a given name and getting the name for a given type. Both need to address name spaces.

**rf\_manager.get\_type\_by\_name()** can be used to get a representation of a type with a given name. If the parameter uses the scope operator, the qualified name's type is returned (or NULL if no such type exists). If the parameter is a simple name, the requested type is determined similar to the resolution process for the interactive scope described in [D.3.2](#). If more than one type with the same unqualified name exists with the same priority, NULL is returned.

The interactive context package determines which types have priority for resolution. Therefore, meta-programming code using reflection might behave differently when the context package changes. The context for name resolution has nothing to do with the module in which the call to **get\_type\_by\_name()** actually appears, but only with the context package when the call is executed.

The method **rf\_type.get\_name()** continues returning the type's unqualified name. A method called **rf\_type.get\_qualified\_name()** returns a string with the fully qualified name (in `::` format).

See also [Clause 31](#).



## D.5.2 Coverage callback support

As a part of the coverage API, use of type names is made programmatically as user input (as parameters to methods) or as output (a field set by the service).

Methods that take type names (possibly with wild cards) as parameters are **user\_cover\_struct.scan\_cover()**, **covers.set\_weight()**, and **covers.set\_at\_least()**. The same rules for commands in interactive scope explained in [D.3.2](#) hold here. When a type name without wild cards is used, the usual resolution process of using the priorities with respect to the interactive context package applies. When a wild card is used, the reference is taken to be all types whose name matches the pattern, regardless of the package qualifier.

The field **user\_cover\_struct.struct\_name** can be used to make a type name an output. For compatibility, this field can only be a simple name. A field of **user\_cover\_struct** called **package\_name** holds the name of the package in which the output type was declared. See also [15.7.1](#).

## D.6 Code comparison

This subclause uses example code for three *e* verification components—**vr\_xbus**, **vr\_xserial**, and **vr\_xsoc**—to compare how typical code looks with the *e* naming conventions and how it would be written with name space support (note the **highlighted** differences shown in [Table D.1](#)). The only places where qualified names are required by the compiler are places where they are also needed for the readability of the code (e.g., the scoping operator is needed in the **vr\_xsoc\_\* . e** example).

**Table D.1—Comparing code with and without name spaces**

Typical <i>e</i> code	Modified to use name spaces
<b>vr_xbus_*.e</b>  <pre> package vr_xbus; type vr_xbus_env_name_t : []; unit vr_xbus_env_u {     name      : vr_xbus_env_name_t;     monitor   : vr_xbus_monitor_u }; unit vr_xbus_monitor_u {...} </pre>	<b>vr_xbus_*.e</b>  <pre> package vr_xbus; type env_name_t : []; unit env_u {     name      : env_name_t;     monitor   : monitor_u }; unit monitor_u {...} </pre>
<b>vr_xserial_*.e</b>  <pre> package vr_xserial; type vr_xserial_env_name_t : []; unit vr_xserial_env_u {     name      : vr_xserial_env_name_t;     agent     : vr_xserial_agent_u                 is instance }; unit vr_xserial_agent_u {...} </pre>	<b>vr_xserial_*.e</b>  <pre> package vr_xserial; type env_name_t : []; unit env_u {     name      : env_name_t;     agent     : agent_u is instance }; unit agent_u {...} </pre>
<b>vr_xsoc_*.e</b>  <pre> package vr_xsoc; type vr_xsoc_env_name_t : []; unit vr_xsoc_env_u {     name      :                     vr_xsoc_env_name_t;     xbus_evc   : vr_xbus_env_u;     xserial_A_evc : vr_serial_env_u;     xserial_B_evc : vr_serial_env_u } </pre>	<b>vr_xsoc_*.e</b>  <pre> package vr_xsoc; type env_name_t : []; unit env_u {     name      : env_name_t;     xbus_evc   : vr_xbus::env_u;     // No qualification would result     // in an ambiguity error     xserial_A_evc : vr_serial::env_u;     xserial_B_evc : vr_serial::env_u } </pre>

## Annex E

(informative)

### Reflection API examples

#### E.1 Type information interface

The following is an example of a very simple use of the type information interface. It is the implementation of a method that receives a struct name as parameter, and prints out the fields with their types, and the methods with their parameter and return types.

```
print_struct(name: string) is {
    var s : rf_struct = rf_manager.get_type_by_name(name)
                          .as_a(rf_struct);

    outf("struct - %s\n", s.get_name());

    if s is a rf_like_struct (ls) then {
        outf(" inherits from - %s\n",
            ls.get_supertype().get_name())
    };

    for each (f) in s.get_declared_fields() do {
        outf(" field - %s: %s\n", f.get_name(),
            f.get_type().get_name())
    };

    for each (m) in s.get_declared_methods() do {
        outf(" method - %s()\n", m.get_name());

        for each (p) in m.get_parameters() do {
            outf(" parameter - %s: %s\n", p.get_name(),
                p.get_type().get_name())
        };

        if m.get_result_type() != NULL then {
            outf(" result type - %s\n",
                m.get_result_type().get_name())
        }
    }
}
```

The following two files serve as a trivial design in order to show the output of the *print\_struct* utility (the same code is referenced in the examples in other subclauses).

##### file1.e

```
type size_t : [big, medium, small];

struct packet {
    size : size_t;
    data : int (bits:256);
    foo(id:int, name:string) is empty
}
```

```
};

extend sys {
  packets : list of packet;
  keep packets.size() > 3 and packets.size() < 7
}
```

### file2.e

```
import file1.e;

extend packet {
  corrupt : bool;
  foo(id:int, name:string) is also {};

  bar(): int is empty
}
```

This is the output of running the utility on the preceding code:

```
file2> print_struct("packet")
struct - packet
  inherits from - any_struct
  field - size: size_t
  field - data: int (bits: 256)
  field - corrupt: bool
  method - foo()
    parameter - id: int
    parameter - name: string
  method - bar()
    result type - int
```

## E.2 Aspect information interface

The following code illustrates the way aspect information interface might be used. It is an implementation of a method that prints out the content of modules in terms of the type layers that they add to the overall design, a set of modules that are loaded (compiled) on top of the *e* execution environment.

```

print_user_modules() is {
  for each (m) in rf_manager.get_all_user_modules() do {
    outf("module - %s\n",m.get_name());
    for each (tl) in m.get_type_layers() do {
      if tl is a rf_struct_layer (sl) then {
        outf("struct layer - %s\n",
            sl.get_defined_entity().get_name());

        for each (fd) in sl.get_field_declarations() do {
          outf(" field declaration - %s\n",
              fd.get_defined_entity().get_name())
        };

        for each (ml) in sl.get_method_layers() do {
          outf(" method layer - %s (%s)\n",
              ml.get_defined_entity().get_name(),
              ml.as_a(rf_method_layer).get_method_kind()
              .to_string())
        }
      }
    }
  }
}

```

Here is a possible output of this utility. In this case, it runs on the trivial design from the previous example (see [E.1](#)), namely modules file1.e and file2.e.

```

file2> print_user_modules()
module - file1
struct layer - packet
    field declaration - size
    field declaration - data
    method layer - foo (is)
struct layer - sys
    field declaration - packets
module - file2
struct layer - packet
    field declaration - corrupt
    method layer - foo (also)
    method layer - bar (is)

```

### E.3 Value query interface

It is hard to find an intuitive use for the object query interface that is simple enough to serve as an example. The following code is a very simple utility that prints out the state of objects recursively. It is somewhat artificial, since it prints out only enumerated and Boolean fields.

```

print_struct_recursive(obj: any_struct) is {
  var s : rf_struct = rf_manager.get_struct_of_instance(obj);

  outf("instance of %s\n",s.get_name());

```

```

for each (f) in s.get_fields() do {
  if f.get_declaration().get_module().is_user_module() then {
    outf("field %s - ",f.get_name());
    var vh : rf_value_holder = f.get_value(obj);

    if vh.get_type() is a rf_scalar (e) then {
      outf("%s",vh.get_type().
        value_to_string(vh.get_value().unsafe()))

    } else if vh.get_type() is a rf_struct (s) then {
      print_struct_recursive(vh.get_value().unsafe())

    } else if vh.get_type() is a rf_list (l) and
    l.get_element_type() is a rf_struct {
      outf("\n");
      var size : int = rf_manager.get_list_size(vh.get_value());
      for i from 0 to size-1 do {
        outf("%d: ",i);
        print_struct_recursive(rf_manager.
          get_list_element(vh,i).get_value().unsafe())
      }
    };
    outf("\n")
  }
}
}

```

Here is the result of calling this method, given the little design of the previous examples (see [E.1](#) modules file1.e and file2.e).

```

file2> gen -seed = 5
Doing setup ...
Generating the test using seed 5...
file2> print_struct_recursive(sys)
instance of sys
field packets -
0: instance of packet
field size - small
field data - -5319061515555392341
field corrupt - TRUE
1: instance of packet
field size - small
field data - 3230878320328792872
field corrupt - FALSE
2: instance of packet
field size - medium
field data - 2775930122720983980
field corrupt - TRUE
3: instance of packet
field size - big
field data - 2044827916054152830
field corrupt - FALSE

```

## Annex F

(informative)

### Encryption Targets

Deciding on which parts of an IP to encrypt (encryption targets) can have a significant impact on the IP provider and the end-user, in terms of integrating, supporting and extending the IP. The guidelines set forth in this section aim to minimize this overhead while providing for reasonable IP protection.

#### F.1 Files

Code that is part of an IP may be encrypted when the code is:

- a) Considered stable
- b) Complete and self-sufficient
- c) Considered to have commercial or business value.

It is strongly encouraged to use the following recommendations for encrypted files:

- a) A non-encrypted header file should accompany each encrypted file to publish its external and internal interface, identify important constructs and indicate their access permissions.
- b) Encrypted files should include debugging aids such as assertions and messaging constructs to provide information about the internal state.

#### F.2 IP Components

A group of files may define standard IP components such as a Bus Functional Model (BFM) or a sequence library. Certain components can greatly affect the interaction of the end-user with the IP and therefore, additional recommendations should be taken into consideration, as shown in [Table 46](#).

**Table 46—Encryption recommendations for IP components**

<b>Architectural Model</b>	Can be left unencrypted as it is helpful to the end-user to see the overall structure of the IP.
<b>Bus Functional Model</b>	Can safely be encrypted as it is unlikely to require significant extensions or debugging once stable.
<b>Sequence Library</b>	Can be left unencrypted for the end-user to use and extend in tests.
<b>Checkers</b>	Can safely be encrypted apart from any associated events or method prototypes that the end-user may use to extend them.
<b>Coverage Models</b>	Can safely be encrypted apart from the coverage groups and important coverage items that the end-user may use to extend coverage information.
<b>Constraints</b>	Can be left unencrypted to facilitate generation analysis and debugging.

