2. *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.

2.1 Lexical conventions

The following sections describe the lexical conventions of ***e***.

2.1.1 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.

2.1.2 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.

2.1.3 Comments and white space

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

Comments within code segments can be marked with double dashes (**--**) or double slashes (**//**):

```
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;
'>
```

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.1.4 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
- MVL literals
- Predefined constants
- Literal string
- Literal character

2.1.4.1 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 3.1.2.3).

The notations shown in Table 1 can be used to represent unsized numbers.

Table 1—Representing unsized numbers in expressions

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

2.1.4.2 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*

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

where *width-number* is a decimal integer specifying the width of the literal in bits and *value-number* is the value of the literal and can be specified in one of four radixes, as shown in Table 2.

124

Table 2—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 are ignored. If *value-number* is less that the specified size, it is padded by zeros (`0`).

2.1.4.3 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".

The mvl type is a superset of the capabilities provided by the `@x` and `@z` syntax allowed in HDL tick notation. 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} ; -- HDL tick notation is 'sig@x' = 0x3
```

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, n.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.1.4.3.1 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, 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. A most significant bit (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 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 ('data' = 32'bx), only the 4-value subset is supported (0, 1, u, 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}.

2.1.4.3.2 Examples

```
32'hffffxxxx
32'HFFFFXXXX
//16'_b1100uuuuu     --illegal because (_) is between (‘) and base
19'oL0001
14'D123     -- illegal because decimal literals are not supported
64'bz_1111_0000_1111_0000
```

2.1.4.3.3 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., `l.add(32'b0)`

2.1.4.4 Predefined constants

The set of *e* *predefined constants* is shown in Table 3.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

1 5 10 129 15 20 25 30 35 40 45 50 55

Table 3—Predefined constants

Constant	Description
`TRUE`	For Boolean variables and expressions.
`FALSE`	For Boolean variables and expressions.
`NULL`	For structs, this specifies a NULL pointer. For character strings, this specifies an empty string.
`UNDEF`	UNDEF signifies NONE where an index is expected. UNDEF has the value -1.
`MAX_INT`	Represents the largest 32-bit **int** (2^{31} -1)
`MIN_INT`	Represents the smallest 32-bit **int** (-2^{31}).
`MAX_UINT`	Represents the largest 32-bit **uint** (2^{32}-1).

2.1.4.5 Literal string

A *literal string* is a sequence of zero or more ASCII characters enclosed by double quotes (“ ”). The escape sequences shown in Table 4 can also be used to specified special characters, e.g., a tab (`\t`).

Table 4—Escape sequences in strings

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

2.1.4.6 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 literal character shown below 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.

2.1.5 Names, keywords, and macros

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

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.1.5.1 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).

2.1.5.2 e file names

The syntax of an ***e*** module name (a file name) is the same as the syntax of UNIX file names, except:

— ‘@’ and ‘~’ are not allowed as the first character of a file name.
— ‘[‘, ‘]’, ‘{‘, ‘}’ are not allowed in file names.
— Only one ‘.’ is allowed in a file name.

NOTE—Many ASCII characters are not handled correctly when used within file names in some UNIX commands. These characters include control characters, spaces, and the characters reserved for command line parsing, such as ‘-’, ‘|’, and ‘<‘.

2.1.5.3 *e* keywords

The keywords listed in Table 5 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***)**”, “before” in “**keep gen** *x* **before** *y*”, or “computed” in “**define *def* as computed**”.

Table 5—Keywords

all of		and	as_a	assert
assume	~~async~~	attribute	before	bit
bits	bool	break	byte	bytes
c export	case	change	check that	
compute	computed	consume	continue	cover
cross	~~evl call~~	~~evl callback~~	~~evl method~~	cycle
default	define	delay	detach	do
down to		each	edges	else
emit	event	exec	expect	extend
fail	fall	file	first of	for
force	from	gen	global	
if	#ifdef	#ifndef	in	index
int	is	is a	is also	is c routine
is empty	is first	is inline	is instance	is not a
is not empty	is only	is undefined	item	keep
keeping	key	like	line	list of
matching	me	nand	new	nor
not	not in	now	on	only

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Table 5—Keywords (Continued)

or		**pass**	**prev**	**print**
range	**ranges**	**release**	**repeat**	**return**
reverse	**rise**	**routine**	**select**	**session**
soft	**start**	**state machine**	**step**	**struct**
string	**sync**	**sys**	**that**	**then**
time	**to**	**transition**	**true**	**try**
type	**uint**	**unit**	**until**	**using**
var	**verilog code**	**verilog function**	**verilog import**	**verilog simulator**
verilog task	**verilog time**	**verilog timescale**	**verilog trace**	**verilog variable**
vhdl code	**vhdl driver**	**vhdl function**	**vhdl procedure**	~~**vhdl driver**~~
vhdl simulator	**vhdl time**	**when**	**while**	**with**

Verify which keywords to remove from this list (and the text in general) [and reorder list]

2.1.5.4 Macros

e macros (created with the **define** statement) can be defined with or without an initial ` character. There are two important characteristics of ***e*** macros defined with an initial ` character.

— They share the same name space as Verilog macros.
— The ` character shall be included when referencing the macro name.

2.1.5.5 String matching pseudo-variables

A successful match **in what?? 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 2.10.

**Why add this here [see Issue #196]; it appears to duplicate section 2.9.4??

2.2 Syntactic elements

Every ***e*** construct belongs to a construct category which determines how the construct can be used. There are five categories of ***e*** constructs, as shown in Table 6.

Table 6—Construct categories

Category	Description
Statements	Statements are top-level constructs and are valid within the begin-code <' and end-code '> markers. See 2.2.1 for a list and brief description of ***e*** statements.
Struct members	Struct members are second-level constructs and are valid only within a struct definition. See 2.2.2 for a list and brief description of ***e*** struct members.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Table 6—Construct categories (Continued)

Category	Description
Actions	Actions are third-level constructs and are valid only when associated with a struct member, such as a method or an event. See 2.2.3 for a list and brief description of ***e*** actions.
Expressions	Expressions are lower-level constructs that can be used only within another ***e*** construct. See 2.2.4 for a list and brief description of ***e*** expressions.
Ranges	Ranges are **xxx-level?? constructs that can be used in specific contexts to specify a range of valid values. See 2.2.5 for a list and brief description of ***e*** ranges.

The syntax hierarchy roughly corresponds to the level of indentation shown below:

```
statements
    struct members
        actions
            expressions
**ranges??
```

2.2.1 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;
    import cpu_test0012;
'>
```

In general, within a given ***e*** module, statements can appear in any order except **import** statements shall appear before any other statements and only **verilog import** statements, preprocessor directives, or any defines (**#ifdef**, **#ifndef**, **define**, **define as**, **define as computed**) can precede **import** statements. See 21.2 for an example of a special case where this restriction also applies to **import** statements in different ***e*** modules.

Table 7 shows the complete list of ***e*** statements.

Table 7—Statements

Statement	Description
`struct`	Defines a new data structure (see 4.2).
`type`	Defines an enumerated data type or scalar subtype (see 3.7.1, 3.7.2, or 3.7.3).
`extend`	Modifies a previously defined struct or type (see 4.3 or 3.7.4).
`define`	Extends the ***e*** language by defining new commands, actions, expressions, or any other syntactic element (see 13.1, or 13.2).

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Table 7—Statements (Continued)

Statement	Description
`#ifdef, #ifndef`	Used together with define statements to place conditions on the *e* parser (see 20.1).
`import`	Reads in an *e* file (see 21.2).
`unit`	Defines a data struct associated with an HDL component or block (see 5.2.1).
`verilog import`	Reads in Verilog macro definitions from a file (see 25.1.3).
`verilog code`	Writes Verilog code to the stubs file, which is used to interface *e* programs with a Verilog simulator (see 25.1.1).
`verilog time`	Specifies the Verilog simulator time resolution (see 25.1.5).
`verilog variable reg | wire`	Specifies a Verilog register or wire to drive from *e* (see 25.1.6).
`verilog variable memory`	Specifies a Verilog memory to access from *e* (see 25.1.7).
`verilog function`	Specifies a Verilog function to call from *e* (see 25.1.2).
`verilog task`	Specifies a Verilog task to call from *e* (see 25.1.4).
`vhdl code`	Writes VHDL code to the stubs file, which is used to interface *e* programs with a VHDL simulator (see 25.2.1).
`vhdl driver`	Used to drive a VHDL signal continuously via the resolution function (see 25.2.2).
`vhdl function`	Declares a VHDL function defined in a VHDL package (see 25.2.3).
`vhdl procedure`	Declares a VHDL procedure defined in a VHDL package (see 25.2.4).
`vhdl time`	Specifies the VHDL simulator time resolution (see 25.2.5).

Verify which keywords to remove from this list (and the text in general)

2.2.2 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 4.2). They can extend over several lines and are separated by semicolons. For example, the following struct "packet" has two struct members, len and data:

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

A struct can contain multiple struct members of any type in any order. Table 8 gives a brief description of each *e* struct member.

2.2.3 Actions

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

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Table 8—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 test goals and collects data on how well the testing is meeting those goals.
Temporal declaration	Defines *e* events and their associated actions.

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

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 sections highlight particular types of actions.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.2.3.1 Creating or modifying variables

Action	Description
var	Defines a local variable (see 16.2).
=	Assigns or samples values of fields, local variables, or HDL objects (see 16.3).
op=	Performs a complex assignment (such as add and assign, or shift and assign) of a field, local variable, or HDL object (see 16.4).
force	Forces a Verilog net or wire to a specified value, over-riding the value from driven from the DUT (see 25.3.1).
release	Releases the Verilog net or wire that was previously forced (see 25.3.2).

2.2.3.2 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 18.1.1).
case labeled-case-item	Executes one action block out of multiple action blocks depending on the value of a single expression (see 18.1.2).
case bool-case-item	Evaluates a list of Boolean expressions and executes the action block associated with the first expression that is true (see 18.1.3).

2.2.3.3 Executing actions iteratively

Action	Description
while	Executes an action block repeatedly until a Boolean expression becomes FALSE (see 18.2.1).
repeat until	Executes an action block repeatedly until a Boolean expression becomes TRUE (see 18.2.2).
for each in	For each item in a list that is a specified type, executes an action block (see 18.2.3).
for from to	Executes an action block for a specified number of times (see 18.2.4).
for	Executes an action block for a specified number of times (see 18.2.5).
for each line in file	Executes an action block for each line in a file (see 18.3.1).
for each file matching	Executes an action block for each file in the search path (see 18.3.2).

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.2.3.4 Controlling program flow

Action	Description
break	Breaks the execution of the enclosing loop (see 18.4.1).
continue	Stops execution of the enclosing loop and continues with the next iteration of the same loop (see 18.4.2).

2.2.3.5 Invoking methods and routines

Action	Description
method()	Calls a regular method (see 15.2.3).
tcm()	Calls a TCM (see 15.2.1).
start tcm()	Launches a TCM as a new thread (a parallel process) (see 15.2.2).
Calling Predefined Routines: routine()	Calls an ***e*** predefined routine (see 24.10).
compute method()	Calls a value-returning method without using the value returned (see 15.2.4).
return	Returns immediately from the current method to the method that called it (see 15.2.5).

2.2.3.6 Performing time-consuming actions

Action	Description
emit	Causes a named event to occur (see 8.2.2).
sync	Suspends execution of the current TCM until the temporal expression succeeds (see 11.1.1).
wait	Suspends execution of the current time-consuming method until a given temporal expression succeeds (see 11.1.2).
all of	Executes multiple action blocks concurrently, as separate branches of a fork. The action following the **all of** action is reached only when all branches of the **all of** have been fully executed (see 11.2.1).
first of	Executes multiple action blocks concurrently, as separate branches of a fork. The action following the **first of** action is reached when any of the branches in the **first of** has been fully executed. (see 11.2.2)
state machine	Defines a state machine (see 27.2.1).

2.2.3.7 Generating data item

Action	Description
gen	Generates a value for an item, while considering all relevant constraints (see 7.3.1).

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.2.3.8 Detecting and handling errors

Action	Description
check that	Checks the DUT for correct data values (see 14.2.1).
dut_error()	Defines a DUT error message string (see 14.2.3).
assert	Issues an error message if a specified Boolean expression is not true (see 14.4.2).
warning()	Issues a warning message (see 14.3.1).
error()	Issues an error message when a user error is detected (see 14.3.2).
fatal()	Issues an error message, halts all activities, and exits immediately (see 14.3.3).
try	Catches errors and exceptions (see 14.3.4).

2.2.3.9 Printing

Action	Description
print	Prints ***e*** expressions (see 14.1.1).

2.2.4 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:

- a literal value
- a constant
- an ***e*** entity, such as a method, field, list, or struct
- an HDL entity, such as a signal

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

2.2.5 Ranges

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

- field declarations to specify constraints on the generation
- coverage declarations
- case action
- some expressions

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

[*range***, ...]**

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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.

2.3 Struct hierarchy and name resolution

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

2.3.1 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 1 shows the partial hierarchy of a typical ***e*** program. The predefined structs are shown in **bold**.

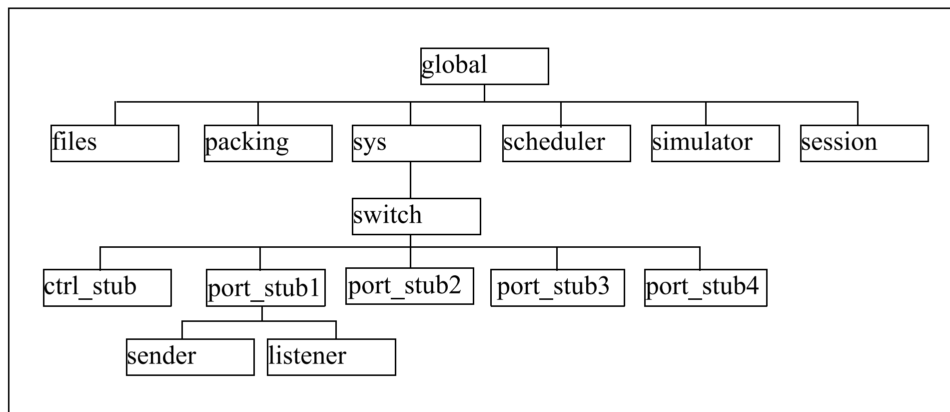

Figure 1—Diagram of struct hierarchy

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

2.3.1.2 Files struct

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

2.3.1.3 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.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.3.1.4 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. See 25.1.5 and 25.2.5 for information on how the timescale is determined.

2.3.1.5 Scheduler struct

The **scheduler** struct contains predefined methods for accessing active TCMs and terminating them.

2.3.1.6 Simulator struct

The **simulator** struct controls the HDL simulator and has a predefined method to access Verilog macros at runtime.

2.3.1.7 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:

- **session.user_time**
- **session.system_time**
- **session.specman_memory** **keep this??
- **session.check_ok**
- **session.events**

48

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

2.3.1.7.1 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");
  };
};
```

2.3.1.7.2 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`.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.3.2 Referencing *e* entities

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

2.3.2.1 Structs and fields

Any user-defined struct can be instantiated as a field of ~~the **sys** struct or of another~~ 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 2.3.3 for more information.
- If the path begins with a period (.), the path is assumed to start with the implicit variable **it**. *See also*: 2.15.3.
- A special syntax is required to reference struct subtypes and fields under struct subtypes (see 3.1.6).

2.3.2.2 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 2.3.4). *See also*: Clause 19, Clause 23, Clause 24, and Clause 26.

Example 1

The example below 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.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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;
};
'>
```

2.3.2.3 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 variable "tmp" below is not specified, it is necessary to use the full name for the enumerated type value "destination'b":

```
m() is {
    var tmp := destination'b;
};
```

2.3.3 Implicit variables

Many *e* constructs create implicit variables. The scope of these implicit variables is the construct that creates them. Two of these implicit variables, **me** and **it**, are used in pathnames when referencing *e* entities.

Except for **result** (see 2.3.3.3), values cannot be assigned to implicit variables. An assignment such as "**me** = packet" shall generate an error.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.3.3.1 it

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

— list pseudo-methods (see Clause 19)
— **for each** (see 18.2.3)
— **gen**… **keeping** (see 7.3.1)
— **keep for each** (see 7.2.4)
— **keep** .**is_all_iterations**() (see 7.2.3
— **new with** (see 2.15.2)
— list with **key** declaration (see 19.8.1)

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**.

2.3.3.2 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.

2.3.3.3 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 15.2.5). 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;
};
```

2.3.3.4 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 19)
— **for each** (see 18.2.3)
— **keep for each** (see 7.2.4)

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;
};
```


Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.3.4 Name resolution rules

The following sections describe how names are resolved, depending on whether they include a path.

2.3.4.1 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**.

2.3.4.2 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 an enumerated type. There cannot be two identical enumerated types.
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 13).
— No user-defined constant can have the same name as a predefined constant (see 2.1.4.4).
— No two enumerated types can have the same *enum-type-name* (see 3.7).

2.4 Operator precedence

Table 9 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 toe compute.

Table 9—Operators in order of precedence

Operator	Operation type
[ ]	List indexing (subscripting) (see 2.11.1)
[..]	List slicing (see 2.11.3)
[:]	Bit slicing (selection) (see 2.11.2)
f(...)	Method and routine calls (see 2.2.3.5)

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Table 9—Operators in order of precedence (Continued)

Operator	Operation type
.	Field selection (see 2.15.3)
~, ! (not)	Bitwise not, Boolean not (see 2.6.1, 2.7.1)
{... ; ...}	List concatenation (see 2.11.4)
var x: list of uint = {1;2;3};%{... , ...} //list of uint	Bit concatenation (see)
Unary + -	Unary plus, minus (see 2.8.1)
*, /, %	Binary multiply, divide, modulus (see 2.8.2)
+, -	Binary add and subtract (see 2.8.2)
>> <<	Shift right, shift left (see 2.6.3)
< <= > >=	Comparison (see 2.9.1)
is [not] a	Subtype identification (see 2.15.1)
== !=	Equality, inequality (see 2.9.2)
=== !==	Verilog four-state comparison (see 2.9.3)
~ !~	String matching (see 2.9.4)
in	Range list operator (see 2.9.5)
&	Bitwise and (see 2.6.2)
|	Bitwise or (see 2.6.2)
^	Bitwise xor (see 2.6.2)
&& (and)	Boolean and (see 2.7.2)
|| (or)	Boolean or (see 2.7.3)
=>	Boolean implication (see 2.7.4)
? :	Conditional operator ("a ? b : c" means "if a then b else c") (see 2.15.5)

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

See Also: Clause 3 for information on the precision of operations and assignment rules.

2.5 Evaluation order of expressions

In e, it is defined that "and" (&&) and "or" (||) use left-to-right lazy evaluation. Consider the following statement:

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


Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

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

Take for example the following statement:

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

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

NOTE—The left-to-right evaluation implemented in *e* assures predictable results, but that order is not guaranteed for other compilers. Avoid writing code that depends on the evaluation order.

2.6 Bitwise operators

The following sections describe the *e* bitwise operators. *See also*: 24.3.2 and 25.4.1.

2.6.1 ~

Purpose	Unary bitwise negation	
Category	Expression	
Syntax	*~exp*	
Parameters	*~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;
```

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.6.2 & | ^

<table>
<tr><td>Purpose</td><td colspan="2">Binary bitwise operations</td></tr>
<tr><td>Category</td><td colspan="2">Expression</td></tr>
<tr><td>Syntax</td><td colspan="2">exp1 operator exp2</td></tr>
<tr><td rowspan="2">Parameters</td><td>exp1, exp2</td><td>A numeric expression or an HDL pathname.</td></tr>
<tr><td>operator</td><td>operator is one of the following:
& performs an AND operation.
| performs an OR operation.
^ preforms and XOR operation.</td></tr>
</table>

This performs an AND, OR, or XOR of both operands, bit by bit.

Syntax example:

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

2.6.3 >> <<

<table>
<tr><td>Purpose</td><td colspan="2">Shift bits left or right</td></tr>
<tr><td>Category</td><td colspan="2">Expression</td></tr>
<tr><td>Syntax</td><td colspan="2">exp1 operator exp2</td></tr>
<tr><td rowspan="3">Parameters</td><td>exp1</td><td>A numeric expression or an HDL pathname.</td></tr>
<tr><td>operator</td><td>operator is one of the following:
<< performs a shift-left operation.
>> performs a shift-right operation.</td></tr>
<tr><td>exp2</td><td>A numeric expression.</td></tr>
</table>

This shifts each bit of the first expression to the right or to the left 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 most significant bits.

The result of a shift by more than 31 bits is undefined.

Syntax example:

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

2.7 Boolean operators

The following sections describe the ***e*** Boolean operators. *See also*: 25.4.1.

2.7.1 ! (not)

Purpose	Boolean not	
Category	Expression	
Syntax	**!***exp* **not** *exp*	
Parameters	*exp*	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));
```

2.7.2 && (and)

Purpose	Boolean and	
Category	Expression	
Syntax	*exp1* **&&** *exp2* *exp1* **and** *exp2*	
Parameters	*exp1*, *exp2*	A Boolean expression or an HDL pathname.

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

Syntax example:

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

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.7.3 || (or)

Purpose	Boolean or
Category	Expression
Syntax	*exp1* || *exp2* *exp1* **or** *exp2*
Parameters	*exp1*, *exp2* A Boolean expression or an HDL pathname.

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

Syntax example:

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

2.7.4 =>

Purpose	Boolean implication
Category	Expression
Syntax	*exp1* => *exp2*
Parameters	*exp1*, *exp2* 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: 7.2.13.

Syntax example:

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


Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.7.5 now

Purpose	Boolean event check
Category	Boolean expression
Syntax	**now** @*event-name*
Parameters	*event-name* 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 8.

Syntax example:

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

2.8 Arithmetic operators

The following sections describe the ***e*** arithmetic operators (*See also*: 25.4.1).

2.8.1 Unary + -

Purpose	Unary plus and minus
Category	Expression
Syntax	-*exp* +*exp*
Parameters	*exp* 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);
```

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.8.2 + - * / %

Purpose	Binary arithmetic
Category	Expression
Syntax	*exp1 operator exp2*
Parameters	*exp1*, *exp2* — A numeric expression or an HDL pathname.
	operator — *operator* is one of the following: + performs addition. - performs subtraction. * preforms multiplication. / performs division and returns the quotient, rounded down. % performs division and returns the remainder.

This performs binary arithmetic operations (*See also*: 24.2).

Syntax example:

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

2.9 Comparison operators

The following sections describe the ***e*** comparison operators (*See also*: 25.4.1).

2.9.1 < <= > >=

Purpose	Comparison of values
Category	Expression
Syntax	*exp1 operator exp2*
Parameters	*exp1*, *exp2* — A numeric expression or an HDL pathname.
	operator — *operator* is one of the following: < returns TRUE if the first expression is smaller than the second expression. <= returns TRUE if the first expression is not larger than the second expression. > returns TRUE if the first expression is larger than the second expression. >= returns TRUE if the first expression is not smaller than the second expression.

This compares two expressions.

Syntax example:

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

2.9.2 == !=

Purpose	Equality of values	
Category	Expression	
Syntax	*exp1 operator exp2*	
Parameters	*exp1*, *exp2*	A numeric, Boolean, string, list, or struct expression.
	operator	*operator* is one of the following: == returns TRUE if the first expression evaluates to the same value as the second expression. != 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 which 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 2.9.3 for a description of using this operator with HDL pathnames.

Comparison methods for the various data types are listed in Table 10.

Table 10—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.
lists	The lists are compared item by item.
structs	The structs addresses are compared

Syntax example:

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

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.9.3 === !==

Purpose	Verilog-style four-state comparison operators	
Category	Expression	
Syntax	'*HDL-pathname*' [!== | ===] *exp* *exp* [!== | ===] '*HDL-pathname*'	
Parameters	*HDL-pathname*	The full path name of an HDL object, this can also include expressions and composite data. See 25.4.1 for more information.
	!==	Determines identity, as in Verilog. Returns TRUE if the left and right operands have identical values (considering also the x and z values).
	===	Determines non-identity, as in Verilog. Returns TRUE if the left and right operands differ in at least 1 bit (considering also the x and z values).
	!=	Returns TRUE if the left and right operands are equal after translating all x values to 0 and all z values to 1.
	==	Returns TRUE if the left and right operands are non-equal after translating all x values to 0 and all z values to 1.
	exp	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.

— '*HDL-pathname*' = = = *literal-number-with-x-and-z values*
  This expression compares a 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, 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, 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. For example, in 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");};
```


Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

2.9.4 ~ !~

Purpose	String matching	
Category	Expression	
Syntax	***"****string****"*** *operator* ***"****pattern-string****"***	
Parameters	*string*	A legal ***e*** string.
	operator	*operator* 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.
	pattern-string	An AWK-style regular expression or a native ***e*** 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 2.10).

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*: 3.1.10.

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$/";
```


Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.9.5 in

Purpose	Check or constrain a value in a list or range	
Category	Expression	
Syntax	*exp1* **in** *exp2*	
Parameters	*exp1*	When the second expression is a range list, e.g., in a **keep** 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 ([ ]) are used. When the second expression is a list, e.g., in a **check**, then the type of the first expression can be one of the following: — a type that is comparable to the element type of the second expression — a list of type that is comparable to the element type of the second expression. For a list, curly braces ({ }) are used.
	exp2	A list or a range list. A *range list* 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 (for example, 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*: 3.1.8 and 2.12.1.

Syntax example:

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

2.10 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*: 24.5.

2.10.1 Native e string matching

Native *e* string matching is attempted on all patterns that are not enclosed in slashes (/). The *e* style is similar to UNIX filename matching. 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 11.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Table 11—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

Example

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"
```

2.10.2 AWK-style string matching

In AWK-style string matching, the standard AWK 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 special characters such as . * [\ ^ $ +? <>, when those characters are used in the same ways as in UNIX regular expressions (regexp). The + and ? characters can be used in the same ways as in UNIX extended regular expression (egrep).

The Perl shorthand notations shown in Table 12 (each representing a single character) can also be used in AWK-style regular expressions.

Table 12—Perl-style regular expressions supported

Shorthand notation	Meaning
`` ` ``	A shortest match operator: ` (back tick)
`\d`	Digit: [0-9]
`\D`	Non-digit
`\s`	Any white-space single char

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Table 12—Perl-style regular expressions supported (Continued)

Shorthand notation	Meaning
`\S`	Any non-white-space single
`\w`	Word char: [a-z A-Z 0-9 _]
`\W`	Non-word char

Add Perl and UNIX references to the Bibliography annex

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
```

2.11 Extraction and concatenation operators

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

2.11.1 [ ]

Purpose	List index operator	
Category	Expression	
Syntax	*list-exp***[***exp***]**	
Parameters	*list-exp*	An expression that returns a list.
	exp	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 2.11.2). *See also*: Clause 19.

Syntax example:

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


Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.11.2 [:]

Purpose	Select bits or bit slices of an expression	
Category	Expression	
Syntax	*exp*[[*high-exp*]:[*low-exp*][:*slice*]]	
Parameters	*exp*	A numeric expression, an HDL pathname, or an expression returning a list of bit or a list of byte.
	high-exp	A non-negative numeric expression. The high expression has to 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 *exp*. For example, if *exp* is a 32-bit integer and the *slice* is bit, the default value is 32.
	low-exp	A non-negative numeric expression, less than or equal to the high expression. The default value is 0.
	slice	Can be **bit**, **byte**, **int**, or **uint**. The default is **bit**.

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 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 left-hand-side 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;
```

2.11.2.1 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 first print statement below 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];
```

2.11.2.2 Accessing nonexistent bits

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

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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: 17.2.14 and 25.4.1.

2.11.3 [..]

Purpose	List slicing operator	
Category	Expression	
Syntax	*exp***[**[*low-exp*]..[*high-exp*]**]**	
Parameters	*exp*	An expression returning a list or a scalar.
	low-exp	An expression evaluating to a non-negative integer. The default is 0.
	high-exp	An expression evaluating to a non-negative integer. The default is the expression size on 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:

a) A list slice of the form a[m..n] requires that n>=m>=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>=0 and m<=a.size(). The size of the slice in this case is a.size()-m.
c) 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.
d) 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.
e) This operator is not supported for unbounded integers.

These rules are also true for the case of list slicing a numeric value. *See also*: 25.4.1.

Syntax example:

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


Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

1 5 10 15 20 25 30 35 40 45 50 55

2.11.4 {... ; ...}

Purpose	List concatenation	
Category	Expression	
Syntax	{*exp*; ...}	
Parameters	*exp*	Any legal ***e*** 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. *See also*: 3.1.8.

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

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
```

2.11.5 %{... , ...}

Purpose	Bit concatenation operator	
Category	Expression	
Syntax	%{*exp1*, *exp2*, ...}	
Parameters	*exp1*, *exp2*	Expressions that receive lists of bits (when on the left-hand side of an assignment operator), or supply lists of bits (when on the right-hand side 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*: 3.3 and Clause 17.

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

- *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;
```

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

2.12 Scalar modifiers

A scalar subtype can be created by using a scalar modifier to specify ~~the range or~~ bit width of a scalar type. ~~The following sections describe the scalar modifiers.~~

~~2.12.1 [range,...]~~

Purpose	Range modifier
Category	Expression
Syntax	**[***range, ...***]**
Parameters	*range* — 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.~~

~~Syntax example:~~

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

Restructure and renumber this section as needed

2.12.2 (bits | bytes : width-exp)

Purpose	Define a sized scalar
Category	Expression
Syntax	**(bits|bytes**: *width-exp***)**
Parameters	*width-exp* — A positive constant expression.

This 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 “word” and “address” types have a bit width of 16.

Syntax example:

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

2.13 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 manual. 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 manual.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Parentheses are also required to invoke a method, pseudo-method, or routine in the following sections: 2.14, 24.10, and 15.2, respectively.

2.14 list.method()

Purpose	Execute list pseudo-method	
Category	Expression	
Syntax	*list-exp*. *list-method***(**[*param*,]**)**...	
Parameters	*list-exp*	An expression that returns a list.
	list-method	One of the list pseudo-methods described in Clause 19.

This executes a list pseudo-method on the specified list expression, item-by-item. 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();
```

2.15 Special-purpose operators

The following special purpose operators are supported.

2.15.1 is [not] a

Purpose	Identify the subtype of a struct instance	
Category	Boolean expression	
Syntax	*struct-exp* **is a** *subtype* [**(***name***)**] *struct-exp* **is not a** *subtype*	
Parameters	*struct-exp*	An expression that returns a struct.
	subtype	A subtype of the specified struct type.
	name	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, then 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.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

— 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: 3.8.1.

Syntax example:

```
if me is a long packet (l) {
    print l;
};
if me is not a long packet {
    print kind;
};
```

2.15.2 new

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

This creates a new struct as follows.

a) Allocate space for the struct.
b) Assign default values to struct fields.
c) Invoke 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.
d) Invoke the **run()** method for the struct, unless the **new** expression is in a construct that is executed before the run phase. For example, if **new** is used in an extension to **sys.init()**, then the **run()** method is not invoked.
e) Execute 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. For example, if the new struct is assigned to a variable of type packet, the new struct will be 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: 23.1.1 and 23.1.2.

Syntax example:

```
var q : packet = new good large packet;
```


Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

1 5 10 15 20 25 30 35 40 45 50 55

2.15.3 .

168 169

Purpose	Refer to fields in structs	
Category	Expression	
Syntax	[[*struct-exp*].] *field-name* [[*struct-exp*].] *event-name* [[*struct-exp*].] *method-name*	
Parameters	*struct-exp*	An expression that returns a struct or a list of structs.
	field-name	The name of the field to reference.
	event-name	The name of the event to reference.
	method-name	The name of the method to reference.

170

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 2.3).

When the struct expression is a list of structs, the expression cannot appear on the left-hand side of an assignment operator.

Syntax example:

```
keep soft port.sender.cell.u == 0xff;
```

2.15.4 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 in the end-code marker **'>**
— in sized numbers (e.g., 2'b11)
— in MVL literals (e.g., 2'bxx).

171

See also: Clause 13 and 3.1.6.

Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

1
5
10
15
20
25
30
35
40
45
50
55

2.15.5 ? :

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

172

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

See also: 18.1.

Syntax example:

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


Copyright © 2005 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.