



# *e* Language Field Guide



---

**Syntax Conventions**

---

**Comments, Identifiers & Literals**

---

**Types, Constants, Fields & Variables**

---

**Operators, Loops and Flow Control**

---

**Structs & Units**

---

**Methods & TCMs**

---

**Checking & Coverage**

---

**Packing & Simulator Interface**

---

**Events & Temporal Expressions (TE)**

---

**Generation & Constraints**

---

**Lists & List Pseudo-Methods**

---

**Actions (Predefined Routines)**

---


**Miscellaneous & Specman Test Flow**

---

# *e* Language Field Guide

Revision 1.0 (for *e* v3.3)

The following conventions are used in this guide:

<b>bold</b>	->	Keywords & required syntax elements
<i>name</i>	->	User supplied/specified name
<i>expr</i>	->	User supplied/specified <i>e</i> expression
[xyz]	->	Optional elements ([ ] not included)
[ ]	->	Required elements ([ ] only)
	->	Choice
...	->	Optional repeated element
	->	Explanatory note



## Syntax Conventions



**Comments, Identifiers & Literals**

**Types, Constants, Fields & Variables**

**Operators, Loops and Flow Control**

**Structs & Units**

**Methods & TCMs**

**Checking & Coverage**

**Packing & Simulator Interface**

**Events & Temporal Expressions (TE)**

**Generation & Constraints**

**Lists & List Pseudo-Methods**

**Actions (Predefined Routines)**

**Miscellaneous & Specman Test Flow**

## Comments, Identifiers, & Literals

### Comments

Code blocks are enclosed by `<' >`

`//` or `--` used for inline comments

### Identifiers

Case sensitive

Start with letter, then any combination of a-z, A-Z, “\_” and 0-9

File names follow these same rules and end with “e”

### Unsigned Numeric Literals

All may include “\_” for readability and “-” for negation

Decimals may use K (kilo \*1024) and M (mega \*1024<sup>2</sup>)

	Legal Characters	Examples
Decimal	0..9	2K, 11, 3_456, -8763
Binary	<b>0b</b> , <b>0B</b> then: 0..1	-0b11, 0B10_01
Hex	<b>0x</b> , <b>0X</b> , <b>0h</b> , <b>0H</b> then: 0..9, a..f, A..F	0x2F, -0h10, 0X2a_3b
Octal	<b>0o</b> , <b>0O</b> then 0..7	0o76, -0O37, 0o55_66

### Sized Numeric Literals

All can include “\_” for readability and “-” for negation

	Legal Characters	Examples
Decimal	<b>#d</b> , <b>#D</b> then: 0..9	2'd3, 8'D11, -14'D8763
Binary	<b>#b</b> , <b>#B</b> then: 0..1	-2'b11, 4'b10_01
Hex	<b>#x</b> , <b>#X</b> , <b>#h</b> , <b>#H</b> then: 0..9, a..f, A..F	8'x2F, -4'h1, 16'X2a_3b
Octal	<b>#o</b> , <b>#O</b> then 0..7	3'o7, -6'O37, 12'o55_66

### String Literals

Any ASCII character or escape sequence enclosed in “ ”

Escape sequences: `\n` = new line `\t` = tab

`\f` = form feed `\` = quote

`\\` = backslash `\r` = carriage return

### Character Literals

Single ASCII character enclosed in “ ” & preceded by “0c”

`var c: uint (bytes:2) = 0c“A”`



## Comments, Identifiers & Literals



## Types, Constants, Fields & Variables

## Operators, Loops & Flow Control

## Structs & Units

## Methods & TCMs

## Checking & Coverage

## Packing & Simulator Interface

## Events & Temporal Expressions (TE)

## Generation & Constraints

## Lists & List Pseudo-Methods

## Actions (Predefined Routines)

## Miscellaneous & Specman Test Flow

## Types, Constants, Fields & Variables

### Predefined Types

Type	Description	Default
<b>int</b>	32 bit signed integer	0
<b>uint</b>	32 bit unsigned integer	0
<b>bit</b>	1 bit unsigned integer (values: 0 or 1)	0
<b>byte</b>	8 bit unsigned integer (values: 0..255)	0
<b>time</b>	64 bit unsigned integer (values: 0..2 <sup>63</sup> -1)	0
<b>bool</b>	1 bit boolean (values: 0=FALSE, 1=TRUE)	FALSE
<b>string</b>	A group of ASCII characters enclosed in “ ”	NULL
<b>locker</b>	Struct to control access to shared resources	NULL
<b>file</b>	Used for file I/O	NULL

### User-Defined Types (Statements)

Any user-defined struct/unit can be used as a type.

New scalar types can be defined and extended using:

```
type type_name : [name [=n],...];
```

```
extend type_name : [name [=n],...];
```

New scalar sub-types can be created using:

```
type type_name : [uint (bits:n | bytes:n)];
```

### Type Conversion (Actions)

```
name = expr.as_a(type);
```

### Constants

Name	Description & Usage
<b>TRUE</b>	Used with boolean variables, fields, and expr
<b>FALSE</b>	Used with boolean variables, fields, and expr
<b>NULL</b>	With structs = null pointer, with strings = empty
<b>UNDEF</b>	Indicates NONE when index or value expected
<b>MAX_INT</b>	Largest 32 bit int (2 <sup>31</sup> -1)
<b>MAX_UINT</b>	Largest 32 bit uint (2 <sup>32</sup> -1)
<b>MIN_INT</b>	Smallest 32 bit int (-2 <sup>31</sup> )

### Fields (Struct/Unit Members)

```
[!][%]field_name : type; // !=don't gen %=physical
```

```
[!][%]field_name : type [ (bits:n | bytes:n)]; // limit size
```

```
[!][%]field_name : type [ [min..max ] ]; // constrain gen value
```

```
[!][%]field_name[gen_size] : list of type; // create list
```

### Variables (Actions)

```
var var_name : [ type ] [ = expr ]; // create a variable
```

```
var_name = expr; // assign to existing variable
```



Types, Constants, Fields & Variables

Operators, Loops & Flow Control

Structs & Units

Methods & TCMs

Checking & Coverage

Packing & Simulator Interface

Events & Temporal Expressions (TE)

Generation & Constraints

Lists & List Pseudo-Methods

Actions (Predefined Routines)

Miscellaneous & Specman Test Flow

# Operators, Loops & Flow Control

## Operators and Precedence (Expressions)

Operator	Description
[ ]	List indexing
[..]	List slicing
[:]	Bit slicing
f()	Method call
.	Field selection
~	Bitwise not
!, not	Boolean not
{;}	List concatenation
%{ }	Bit concatenation
Unary +, -	Unary plus, minus
*, /, %	Multiply, divide, modulus
+, -	Add and subtract
>>, <<	Shift right, shift left
<, <=, >, >=	Comparison
is [not] a	Subtype identification
==, !=	Equality, inequality
===, !==	Verilog four-state comparison
~, !~	String matching
[not] in	Range list operator
&,  , ^	Bitwise AND, OR, XOR
&&,   , and, or	Boolean AND, OR
=>	Boolean implication
a ? b : c	Conditional operator (Ex: "if a then b else c")

## Looping Constructs (Actions)

```

while bool_expr [do] { action; ... };
for {initial; bool_expr; step} [do] { action;...};
for iterator from expr [down] to expr [step expr]
  [do] { action;... };
for each [type] [(iterator_name)]
  [using index (index_name)]
  in [reverse] list [do] { action;... };
for each file [(iterator_name)] matching file_name_expr
  [do] { action;... };
for each line [(iterator_name)] in_file file_name
  [do] { action;... };
break; terminates execution of the loop
continue; go to next loop iteration now

```

## Flow Control Constructs (Actions)

```

if bool_expr [then] { action;... }
  [else if bool_expr [then] { action;... } ]
  [else { action;... }];
case {bool_expr[:]} { action;... };
  [default[:] { action;...; }];
case expr {value[:]} { action;... };
  [default[:] { action;...; }];
all of { {action blk1;} {action blk1;}... };
first of { {action blk1;} {action blk1;}... };

```



Operators, Loops & Flow Control

Structs & Units

Methods & TCMs

Checking & Coverage

Packing & Simulator Interface

Events & Temporal Expressions (TE)

Generation & Constraints

Lists & List Pseudo-Methods

Actions (Predefined Routines)

Miscellaneous & Specman Test Flow

## Structs & Units

### Defining (Statements)

```
struct struct_type [like struct_type] {
    struct_member;... };
unit unit_type [like unit_type] { unit_member;... };
```

### Extending (Statements)

```
extend struct_type { new_struct_member;... };
extend unit_type { new_unit_member;... };
```

☞ See also when inheritance below.

### Instantiating (Struct/Unit Members)

```
[!]field_name : struct_type;           // !=do not generate
[!]struct_type;                        // field name=struct type
[!]field_name[gen_size] : list of struct_type; //create a list
name : unit_type is instance;        // instantiate a unit
unit_name is instance;               // field name unit name
field_name[gen_size] : list of unit_type; // create a list
```

```
Example: define, extend & instantiate structs/units
struct my_struct {
    struct_int : int;
};
unit my_unit {
    kind      : [GOOD, BAD];
    unit_int  : int;
};
extend my_struct {
    struct_int2 : int; // Add a new field to my_struct
};
extend sys {
    test_bench : my_unit is instance;
    my_unit is instance; // instance of my_unit
};
```

### when Inheritance (Struct/Unit Members)

```
when value[field] struct_type { struct/unit_member;... };
extend value[field] struct_type { struct/unit_member;... };
```

```
Example of using when inheritance
//Add non-generated field to GOOD version on my_unit
extend GOOD my_unit {
    !data : my_struct;
};
```

### Predefined Methods (Struct/Unit Members)

Common Methods	Description
init()	Initializes ungenerated fields, etc.
pre_generate()	[empty] - Prepares for generation
post_generate()	[empty] - Manipulates generated data
run()	[empty] - Starts TCMs at time 0
extract()	[empty] - Extracts from DUT post test
check()	[empty] - Defines post test check
quit()	Deactivates a struct/unit instance
finalize()	Creates final reports, etc.
copy()	Returns a non-recursive copy of struct
do_print()	Defines print format; called by print action
print_line()	Prints a struct/unit in a single line
do_pack()	Called when struct/unit is packed
do_unpack()	Called when struct/unit is unpacked
hdl_path()	Ties unit to DUT component; units only
get_unit()	Returns reference to a unit
(get try)_enclosing_unit( <i>unit-type</i> )	Rtn ref nearest unit <i>unit-type</i>



## Structs & Units



### Methods & TCMs

### Checking & Coverage

### Packing & Simulator Interface

### Events & Temporal Expressions (TE)

### Generation & Constraints

### Lists & List Pseudo-Methods

### Actions (Predefined Routines)

### Miscellaneous & Specman Test Flow

## Methods & TCMs

### Defining Methods (Struct/Unit Members)

```
method( [input: [*]type, ...] [ :return_type] is {action;...};
```

### Extending Methods (Struct/Unit Members)

```
method( [input: [*]type, ...] [ :return_type]
  is also | is first | is only { action;... };
  ▢ Inputs must match those previously defined.
```

### Defining TCMs (Struct/Unit Members)

```
method( [input: [*]type, ...] [ :return_type] @ event is
  { action;... };
```

### Extending TCMs (Struct/Unit Members)

```
method( [input: [*]type, ...] [ :return_type] @ event
  is also | is first | is only { action;... };
```

### Time Consuming Actions (Actions)

```
sync TE;           // continue when TE succeeds
wait [until] TE;  // wait until next time TE succeeds
```

```
Example: define and extend methods and TCMs
struct test {
  method1() is { out("Hello - m1"); };
  method2() is { out("Hello - m2"); };
  tcm1() @ clk is { wait; out("Hello - tcm1"); };
};
extend test {
  method1() is also { out("2nd msg - m1"); };
  tcm1() is first { wait; out("1st msg - TCM"); };
  method2() is only { out("Only msg - m2"); };
};
Executing method1() results in: Hello - m1; 2nd msg - m1
Executing method2() results in: Only msg - m2
Executing tcm1() results in: 1st msg - TCM; Hello - tcm1
```

### Result & Return Keywords

```
result = expr;
return = [ expr; ]
```

### Using Methods (Actions)

A regular method can be called from a regular method.  
Example: method() is { method1(); method2(); }

A regular method or TCM can be called from a TCM.  
Example: TCM()@event is { method1(); TCM1; }

A TCM can be *started* from a regular method or TCM.  
Example: method() is { method1(); **start** TCM1; }

#### Example: using return, result, & invoking methods

```
extend test {
  data : list of byte;
  !parity1 : byte;
  !parity2 : byte;
  parity_calc1(input:list of byte) :byte is {
    result = 0;
    for each (byte) in input {
      result = result ^ byte
    };
  };
  parity_calc2() :byte is {
    var tmp : byte = 0;
    for each (byte) in data {
      tmp = tmp ^ byte;
    };
    return tmp;
  };
  run() is also {
    start tcm1; //tcm1 defined in prev example
    parity1 = parity_calc1(data);
    parity2 = parity_calc2();
  };
};
```



## Methods & TCMs



### Checking & Coverage

### Packing & Simulator Interface

### Events & Temporal Expressions (TE)

### Generation & Constraints

### Lists & List Pseudo-Methods

### Actions (Predefined Routines)

### Miscellaneous & Specman Test Flow

## Checking & Coverage

### Data Checks (Actions)

```
check [ that ] bool_expr [else dut_error(string-expr)];
```

### Temporal Checks (Struct/Unit Members)

```
expect TE [ else dut_error(string-expr) ];
```

### Setting dut\_error() Response (Actions)

```
set_check ( "match string", Check Effect );
```

Check Effect	Description
<b>ERROR</b>	Print message, increase num_dut_errors, terminate method and test run
<b>ERROR_BREAK_RUN</b>	Print message, increase num_dut_errors, stop simulation at next clock boundary
<b>ERROR_AUTOMATIC</b>	Print message, increase num_dut_errors, terminate method as if hit stop_run()
<b>ERROR_CONTINUE</b>	Print message, increase num_dut_errors, continue the test run
<b>WARNING</b>	Print message, increase num_dut_warnings, continue test run
<b>IGNORE</b>	No message and does not increase error or warning counters

### Coverage Definition (Struct/Unit Members)

```
cover_group_event [using cover_group_options]
is [also] {
item name [ : type=expr ] [using cover_item_options, ...];
cross item1, item2, ... [using cover_item_options, ...];
transition item [using cover_item_options, ...];
};
```

### Coverage Group Options

```
no_collect          count_only          global
text = "string"     weight = uint (default 1)
when = bool_expr   radix = DEC | HEX | BIN (default DEC)
```

### Coverage Item Options

```
no_collect          at_least = uint (default 1)
text = "string"     weight = uint (default 1)
when = bool_expr   radix = DEC | HEX | BIN (default DEC)
ignore | illegal = cover_item_boolean_expr
ranges={range( [n..m], name, [every_count, at_least] );
range( [n..m], name, [every_count, at_least] ); }
```

### Configuring Coverage (Actions)

```
set_config(cover, mode, coverage_mode);
coverage_mode: normal = save each sample/time
count_only = save only count/bucket
```

Typically, **set\_config** is placed in an extension of the global struct setup\_test() method as shown below.

```
Example of defining coverage:
extend packet {
event pkt_data;
cover pkt_data using text="Packet Coverage" is {
item address using illegal=address==0;
item data using
ranges={range([0..100], "sm"),
range([101..255], "lg") },
radix = HEX, at_least = 10 ;
cross address, data ;
};
};
extend global {
setup_test() is also {
set_config(cover, mode, normal);
};
};
```



## Checking & Coverage



### Packing & Simulator Interface

### Events & Temporal Expressions (TE)

### Generation & Constraints

### Lists & List Pseudo-Methods

### Actions (Predefined Routines)

### Miscellaneous & Specman Test Flow

## Packing & Simulator Interface

### Accessing HDL Signals (Expressions)

'[~/HDL][path/]name [ @x | @z ]' // appends to hdl\_path()

### Packing (Expressions)

`dest_name = pack( option, src_name [, src_name, ... ] );`

`unpack( option, src_name, dest_name [, dest_name, ... ] );`

options: **packing.high** **packing.high\_big\_endian**  
**packing.low** **packing.low\_big\_endian**  
**packing.network**

`list_of_bits.swap( unit_size, group_size );`

#### Example of packing and unpacking:

```

struct packet {
    %addr : uint (bits:4); keep addr == 2'b1111;
    %data : uint (bits:8); keep data == 8'b10101010;
};
extend sys {
    pkt1 : packet;
    !pkt2 : packet;
    test() is {
        var packed : list of bit;
        var swapped : list of bit;
        packed=pack(packing.high, pkt1);//1111_10101010
        unpack(packing.low, packed, pkt2); // oops!
        swapped=packed.swap(4,12);
    };
};

```

Executing sys.test() results in:

```

pkt2.addr=2'b1010 and pkt2.data=8'b1111_1010
swapped = 1010_10101111

```

### Verilog Specifics (Statements/Struct/Unit Members)

**verilog import** *filename*;

**verilog code** { "string" [ ; "string" ] };

**verilog time** *unit\_time / precision*;

**verilog variable** '~/HDL/path' using *option*, [ , ... ] ; // reg

options: **net** | **wire**, **forcible**

**drive** = *verilog\_expr*

**hold** = *verilog\_expr* // must also specify drive=

**strobe** = *verilog\_expr*

**verilog variable** '~/HDL/path' [ *mem\_range* ] [ *width* ] ;

**verilog function** '~/HDL/path' ( *param1* [, ...] ) : *rtn\_bit\_size* ;

**verilog task** '~/HDL/path' ( *param1* [, ...] );

### VHDL Specifics (Statements/Struct/Unit Members)

**VHDL code** { "string" [ ; "string" ] } ;

**VHDL driver** '~/HDL/path' using *option* [ , ... ] ;

options: **disconnect value=** *sized\_literal*

**delay=** *numeric\_expr*

**mode=** **INERTIAL** | **TRANSPORT**

**initial value=** *sized\_literal*

**VHDL time** *value units* ;

**VHDL function** 'name' using *option* [ , ... ] ;

options: **interface=** " ( *param1* [, ...] ) return *rtn\_type* "

**library=** " *library\_name* "

**package=** " *package\_name* "

**alias=** " *alias\_name* "

**declarative\_item=** " *VHDL item* ; "

**VHDL procedure** 'name' using *option* [ , ... ] ;

options: same as for VHDL function

### force & release (Actions)

**force** '~/HDL/path' = *expr* ;

**release** '~/HDL/path' ;

☞ The force/release actions work for any VHDL signal, but only work for Verilog signals declared **forcible** in a **verilog variable** statement.



## Packing & Simulator Interface



### Events & Temporal Expressions (TE)

### Generation & Constraints

### Lists & List Pseudo-Methods

### Actions (Predefined Routines)

### Miscellaneous & Specman Test Flow

## Events & Temporal Expressions (TE)

### Defining Events (Struct/Unit Members)

**event** *name*; // named event  
**event** *name is TE*; // temporal event  
**event** *name is only TE*; // extending

### Triggering Events (Actions)

**emit** *event\_name*; // trigger named event (Action)

### Acting On Events (Struct/Unit Members)

**on** *event\_name* { *action*; ... };

TCMs also rely on events (see “Methods & TCMs”).

### Predefined Events

**sys.any**    **sys.tick\_start**    **sys.tick\_end**  
**session.start\_of\_test**    **session.end\_of\_test**  
**struct.quit**    **sys.new\_time**

### Temporal Expressions (Expressions)

TEs are valid only within **wait** and **sync** constructs, **event** declarations, and **expect** constructs.

**@**[*struct\_instance*.]*event\_name* // event as a TE  
**TE**@*sampling\_event* // TE w/ associated sampling event  
**change** | **rise** | **fall** (*expr*)@*sampling\_event* // transition TE  
**change** | **rise** | **fall** ('~/HDL/path')@**sim** // HDL related TE  
**true**(*bool\_expr*)@*sampling\_event* // boolean condition TE  
**cycle**@*sampling\_event* // occurrence of a sampling event  
**delay**(*expr*) // specify a sim time delay of *expr*  
[ *expr* ] [\***TE**] // repetition of TE, *expr*=numeric

[ *min* ] .. [ *max* ] [\***TE**]; **TE2** // “1st match” or TE2 var repeat  
~[ *min* ] .. [ *max* ] [\***TE**] // xlates: [min]\*TE or [min+1]\*TE...  
{ **TE1**; **TE2** [...] } // temporal sequence - TE1 then TE2  
**TE1**=> **TE2** // if TE1 succeeds, then TE2 must occur  
**TE1** and | or **TE2** // logical and/or  
**not TE** // logical negation at any time  
**fail TE** // succeeds when all possibilities to pass are exhausted  
**eventually TE** // TE happens eventually  
**detach**(**TE**) // creates a separate thread  
**TE exec** { *action*; ... } // exec action on TE succeed  
**consume**( @*event* ; // remove event occurrence

#### Some examples of TE:

```
// Define sim controlled event (rise of clk)
event clk is rise ('~/top/clk') @sim;
// Event A occurs if TE1 and TE2 occur in the same
// clk cycle
event A is {TE1 and TE2} @clk;
// Event B occurs if TE2 occurs 1 cycle after TE1
// (action is exec when TE1 occurs)
event B is {TE1 exec{action;} ; TE2} @clk;
// Event C occurs if TE2 occurs within n1 & n2 clk
// cycles after TE1
event C is {TE1; [n1-1..n2-1]; TE2} @clk;
// Event D occurs if TE2 occurs after n clk cycles
// with no TE1
event D is V{[n]* not TE1 ; TE2} @clk;
// Event E occurs if TE3 occurs anytime after TE1
// with no TE2 in between
event E is {TE1; [..]*not TE2; TE3} @clk;

// TE2 must occur 1 clk cycle after TE1
expect TE1 => TE2 @clk;
// If TE1 occurs, eventually TE2 must occur
expect TE1 => (eventually TE2) @clk;
// TE3 must occur n cycles after TE1 or TE2 occur
expect (TE1 or TE2) => { [n-1]; TE3 } @clk;
// If TE1 occurs, TE2 can't occur for n cycles
expect TE1 => { [n]* not TE2; TE2 } @clk;
// If TE1 occurs ~/top/ready must be true
expect (not TE1 or true('~/top/ready'))@clk;
// If TE2 occurs, TE1 must occur n cycles before
expect TE2 => detach( {TE1; [n+1]} )@clk;
```



Events & Temporal Expressions (TE)

Generation & Constraints

Lists & List Pseudo-Methods

Actions (Predefined Routines)

Miscellaneous & Specman Test Flow

## Generation & Constraints

### Generation (Actions)

**gen** *name*; // name can be a field, variable, or struct instance

**gen** *name* **keeping** { [**soft**] *constraint\_expr*; ... };

### Generation Methods (Struct/Unit Members)

Whenever a struct is generated, its predefined methods are run in order. Extend these to tailor generation.

Method	Description
init()	Initializes ungenerated fields, etc.
pre_generate()	[empty] - prepares for generation
post_generate()	[empty] - manipulates generated data

### Constraints (Struct/Unit Members)

[!][%]*field\_name* : *type* [ [ *range* ] ];

**keep** [**soft**] *bool\_expr*;

**keep** [**soft**] *name* [**not**] **in** [ *range* ];

**keep** [**soft**] *name* [**not**] **in** *list*;

**keep** *bool\_expr1* => *bool\_expr2*;

**keep for each** [(*iterator\_name*)

[**using** [**index** (*index\_name*)] [**prev** (*prev\_name*)] ]

**in** *list\_name* { [**soft**] *bool\_expr* | *nested\_for\_each* };

**keep soft** *name* == **select** {*weight* : *value*;

*weight* : *value*;... };

**keep** [**soft**] **gen** (*name1* [, ...] ) **before** (*name2* [, ...]);

**keep** *struct\_list*.**is\_all\_iterations**( .*field\_name* [, ...] );

**keep** *struct\_list*.**is\_a\_permutation**( *list* );

**keep** *name*.**reset\_soft**()

**keep** *unit\_name*.**hdl\_path**() == "[~/HDL/][*path*/]*string*";

**keep** *struct\_name* **is** [**not**] **a** *value*[*field*] *struct\_type*;

#### Constraint Examples:

```
x : int [0..10]; // same as keep x in [0..10]
keep soft x == 1 // x defaults to 1
keep x <= y; // x less than or equal to y
keep x in {3,4,5}; // x must be 3 or 4 or 5
keep x+y == z; // restrict x+y to value z
keep parity = parity_calc(); // force w/ method
```

```
// if x is 1, then y must be in range 1..10
// translates to: x != 1 or y in [0..10]
keep x == 1 => y in [0..10];
```

```
struct packet {
    kind :[good, bad];
    addr : uint;
    data : uint;
};
extend sys {
    pkts : list of packet
    keep pkts.size() < 20; // gen < 20
    keep for each (pkt) in pkts {
        index > 0 =>
            pkt.addr == prev.addr; // same addr
            index==2 =>
                pkt is a good packet; // #3=good
    };
```

```
list1 : list of packet;
!list2 : list of packet;
keep list1.is_all_iterations(.addr);
gen_one() is {
    gen list2 keeping {
        it.is_a_permutation(list1);
    };
};
```



Generation & Constraints



Lists & List Pseudo-Methods

Actions (Predefined Routines)

Miscellaneous & Specman Test Flow

## Lists & List Pseudo-Methods

### Defining Lists (Struct/Unit Members)

```
[![%]field_name[gen_size]:list of type;
[![%]field_name[gen_size]:list (key: field_name) of type;
```

**Ex: To create multi-dim arrays, create list structs:**

```
struct one_dim { a : list of uint; };
struct two_dim { b : list of one_dim; };
```

### Lists Pseudo-Methods (Actions/Expressions)

#### Direct Modification of a List

```
list.add(item | list);           // add item or list to end of list
list.add0(item | list);         // add item or list to start of list
list.clear();                   // delete all items in list
list.delete(index);            // delete item at index (adjust others)
list.fast_delete(index);       // delete item at index (move last)
list.insert(index, item | list); // add item / list at index of list
item = list.pop();             // delete & return last item in list
item = list.pop0();            // delete & return first item in list
list.push(item);               // add item to the end of list
list.push0(item);              // add item to the start of list
list.resize(size [,full, filler, keep_old]); // pad or truncate list
```

#### Queries & General Operations on a List

```
list=list.all(bool_expr);       // all w/ bool_expr=TRUE
list1=list.all_indicies(bool_expr); // all w/ bool_expr=TRUE
list1=list.apply(expr);         // compute on each item
int=list.count(bool_expr);      // # where bool_expr=TRUE
bool=list.exists(index);       // TRUE if index is in the list
item=list.first(bool_expr);     // 1st w/ bool_expr=TRUE
int=list.first_index(bool_expr); // index 1st w/ bool_expr=TRUE
list1=list.get_indicies(list_of_int); // new list index items listed
bool=list.has(bool_expr);      // TRUE if list has bool_expr=TRUE
```

```
bool=list.is_a_permutation(list1); // checks list vs. list1
bool=list.is_empty();              // TRUE if no items in list
item=list.last(bool_expr);        // last item bool_expr=TRUE
int=list.last_index(bool_expr);   // index last w/ bool_expr=TRUE
item=list.max(expr);              // item w/ expr calcs to max
int=list.max_index(expr);         // index w/ expr calcs max
int=list.max_value(expr);         // max value calcd for expr
item=list.min(expr);              // item w/ expr calcs to min
int=list.min_index(expr);         // index w/ expr calcs to min
int=list.min_value(expr);         // min value calcd for expr
list1=list.reverse();             // rev order of items of list
int=list.size();                  // number of items in the list
list1=list.sort(expr);            // sort list incr order of expr value
list1=struct_list.sort_by_field(field); // sort list in incr order
var struct_list_holder := struct_list.split(expr); // split on expr
item=list.top();                  // return last item, no delete
item=list.top0();                 // return first item, no delete
list1=list.unique(expr);          // list1 where expr is unique in list
```

#### Math & Logic Operations on a List

```
bool=list.and_all(bool_expr);    // AND of all w/ bool_expr=TRUE
int=list.average(expr);          // int avg expr calcd for ea item
bool=list.or_all(bool_expr);     // OR of all w/ bool_expr=TRUE
int=list.product(expr);          // int prod of expr calcd ea item
int=list.sum(expr);              // int sum of expr calcd ea item
int=list.crc_8(from, num);        // crc8 bit/byte list-start at from byte
int=list.crc_32(from, num);       // crc32 bit/byte list-start at from byte
int=list.crc_32_flip(from, num); // crc32 ea byte & result "flipped"
```

#### Specifics for Keyed Lists

```
item=list.key(value);            // list item with key = value
int=list.key_index(value);       // index of list item w/ key = value
bool=list.key_exists(value);     // TRUE if key = value exists in list
```



## Lists & List Pseudo-Methods



### Actions (Predefined Routines)

### Miscellaneous & Specman Test Flow

## Actions (Predefined Routines)

### Output (Actions)

**out**(*expr* [, *expr*...]); // output *expr* new line end  
**outf**(*format\_string*, *expr* [, *expr*...]); // formatted output of *exprs*  
*format\_string* = "%[0|-][*min#*][.*max#*](s|d|x|b|o|u)"  
 0 - pads w/ 0s not blanks    s- string        b - binary  
 -- aligns left                d - decimal    o - octal  
                                   x- hex         u - uint

### Arithmetic (Expressions)

*int*=*op*=*value* // (*op* can be: +, -, \*, /, %, &, |, ^) ex: *x* += 1 increments *x* by 1  
*int*=[*min*][*max*](*expr1*, *expr2*); // min / max of *expr1* or *expr2*  
*int*=**abs**(*expr*); // absolute value of *expr*  
*bool*=**odd**(*expr*); // TRUE if *expr* is odd  
*bool*=**even**(*expr*); // TRUE if *expr* is even  
*int*=**ilog**n(*expr1*); // log base-*n* of *expr* (*n*=2, 10)  
*int*=**ipow**(*expr1*, *expr2*); // *x* to the power of *y* (*x*<sup>*y*</sup>)  
*int*=**isqrt**(*expr*); // square root of *expr*  
*int*=**div\_round\_up**(*expr1*, *expr2*); // *expr1* / *expr2* round nxt int

### Bitwise (Expressions)

*bit*=**bitwise\_op**(*expr*); // unary reduct(*op* = and, or, xor, nand, nor, xnor)

### Copy and Compare (Expressions)

*struct\_type*=**deep\_copy**(*struct\_inst*); // recur *struct* copy  
*string\_list*=**deep\_compare**(*struct1*, *struct1*, *max*); // recur all  
*string\_list*=**deep\_compare\_physical**(*struct1*, *struct1*, *max*);

### String Manipulation (Expressions)

*string*=**append**(*expr* [, *expr*...]); // concat *expr* to *string*  
*string*=**appendf**(*expr* [, *expr*...]); // concat *fmt*d (see *outf*)  
*string*=**bin**(*expr* [, *expr*...]); // concat w/ binary num *fmt*  
*string*=**dec**(*expr* [, *expr*...]); // concat w/ dec num *fmt*  
*string*=**hex**(*expr* [, *expr*...]); // concat w/ hex num *fmt*  
*string1*=**quote**(*string*); // copy of *string* in quotes

*string1*=**str\_chop**(*string*, *length*); // truncate *string* to *length*  
*bool*=**str\_empty**(*string*); // TRUE if *string* inited or empty  
*string1*=**str\_exactly**(*string*, *length*); // pad or trunc *string* to *len*  
*string1*=**str\_expand\_dots**(*string*); // use with macro defines  
*string1*=**str\_insensitive**(*reg\_exp*); // return AWK style *reg\_exp*  
*string1*=**str\_join**(*string\_list*, *sep*); // concat using separator  
*int*=**str\_len**(*string*); // get the *string* length  
*string1*=**str\_lower**(*string*); // conv *string* to lower case  
*bool*=**str\_match**(*string*, *reg\_exp*); // TRUE if *string* matches *reg\_exp*  
*string1*=**str\_pad**(*string*, *length*); // pad *string* to *length*  
*string1*=**str\_replace**(*string*, *reg\_exp*, *replacement*);  
*string\_list*=**str\_split**(*string*, *sep*); // break to *strings* on separator  
*string\_list*=**str\_split\_all**(*string*, *sep*); // separator also is *string*  
*string1*=**str\_sub**(*string*, *from\_int*, *length*); // cut out sub*string*  
*string1*=**str\_upper**(*string*); // convert *string* to upper case  
*string1*=**expr.to\_string**(); // convert *expr* (*struct*, *list*, ...) to *string*

### Misc (Actions/Expressions)

#### Actions

**set\_config**(*category*, *option*, *value* [*option*, *value*])  
 [ [**with**] {*action*;...} ]; // set config for only specified actions  
*value\_type*=**get\_config**(*category*, *option*); // value of *option*  
**write\_config**(" *filename* "); // write config to *file.ecfg*  
**read\_config**(" *filename* "); // read config from *file.ecfg*  
**set\_keep**(TRUE | FALSE); // sets keep-across-restore  
*bool*=**get\_keep**(); // gets keep-across-restore  
 ⓘ The keep-across-restore value controls whether config & debug settings are retained across restores and reloads.  
**specman**(" *command*" [,... ] ); // concat & send to Specman  
**spawn**(" *command*" [,... ] ); // concat & exec via *system*()  
**spawn\_check**(" *command*" ); // send cmd-errors = error()

#### Expressions

*int*=**system**(" *command* "); // snd cmd return result code  
*string\_list*=**output\_from**(" *command* "); // get result as *string* list  
*string\_list*=**output\_from\_check**(" *command* "); // calls *error*()  
*string*=**get\_symbol**(" *env\_variable* "); // gets unix *env* variable  
*string*=**date\_time**(); // gets current date and time



Actions (Predefined Routines)



Miscellaneous & Specman Test Flow

## Miscellaneous

### Compile Time Directives

```
import filename;           // load e file; if no ext, "e" used
#ifdef [ ]macro then { e code }; // if macro def, incl e code
#ifndef [ ]macro then { e code }; // if macro not def, incl e code
```

### Macros

```
define [ ]macro replacement; // simple macro replacement
define <macro'non-terminal-type> "match_string"
  as {replacement using <parsing elements>};
define <macro'non-terminal-type> "match_string"
  as computed {replacement using <parsing elements>};
```

### Sharing Resources

```
resource_name: locker; // define locker filed in shared rescr
resource_name.lock(); // reserve resource (waits until avail)
resource_name.unlock(); // release the resource
```

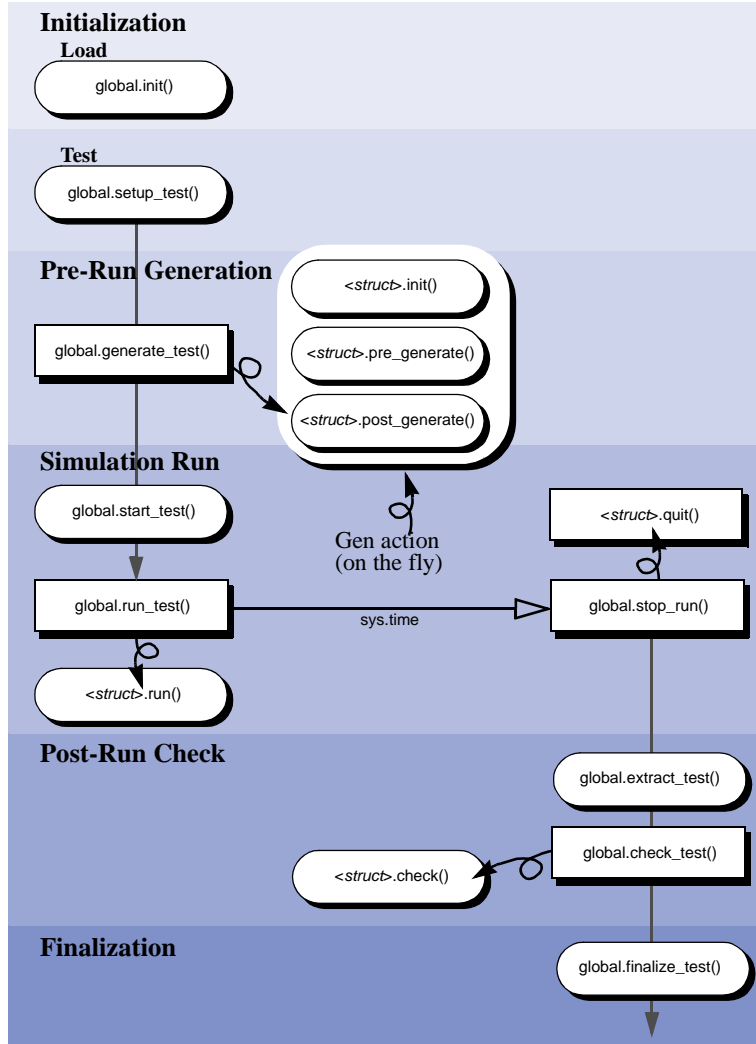
### State Machines

```
state machine state_holder [until final_state]
{ (transition | state) {action;...} ... };
transition can be: state => state or * => state
```

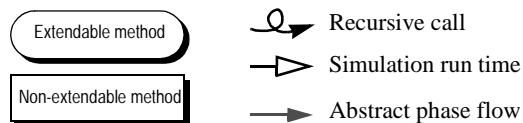
### File I/O

```
file=files.open("filename", "mode", "description");
mode can be: "r", "w", "rw", "a"
bool=files.read[write](file, string); // r / w line as text
bool=files.read[write]_lob(file, string); // r / w line as bin
files.close(file); // close open file
files.flush(file); // flush file buffers
string=files.add_file_type(name,ext,check_if_exists);
```

## Specman Test Flow (Reference)



### Legend



# Build your know-how.

Learn more than just **e** syntax. Learn how to apply the language as a power tool for verifying today's ultra complex systems.

Qualis offers a family of methodology-based training courses focused on Specman Elite™. You'll go from learning the basics of **e** to learning an advanced verification methodology applying random constrained test generation, data/temporal checking, and functional coverage. Detailed course descriptions are available on our website at [www.qualis.com/learning.catalog.html/](http://www.qualis.com/learning.catalog.html/).

Attend a public course in North America or Europe, or schedule a private onsite class. Start building your verification knowledge today.

## Qualis Design Corporation

### *Solutions in Advanced Verification and Design*

Other quick reference cards available in print or from the web: VHDL, Verilog, 1164 Packages and **e** Language.

Phone: +1.503.670.7200

FAX: +1.503.670.0809

Email: [info@qualis.com](mailto:info@qualis.com)

Web: <http://www.qualis.com>

We welcome your comments about this document.

Send them to [info@qualis.com](mailto:info@qualis.com). Identify the revision (Rev 1.0).

© 2001 Qualis Design Corporation. Permission to reproduce and distribute strictly verbatim copies of this document in whole is granted.

