Specman GUTS Course
(Get Up To Speed)

Table of Contents

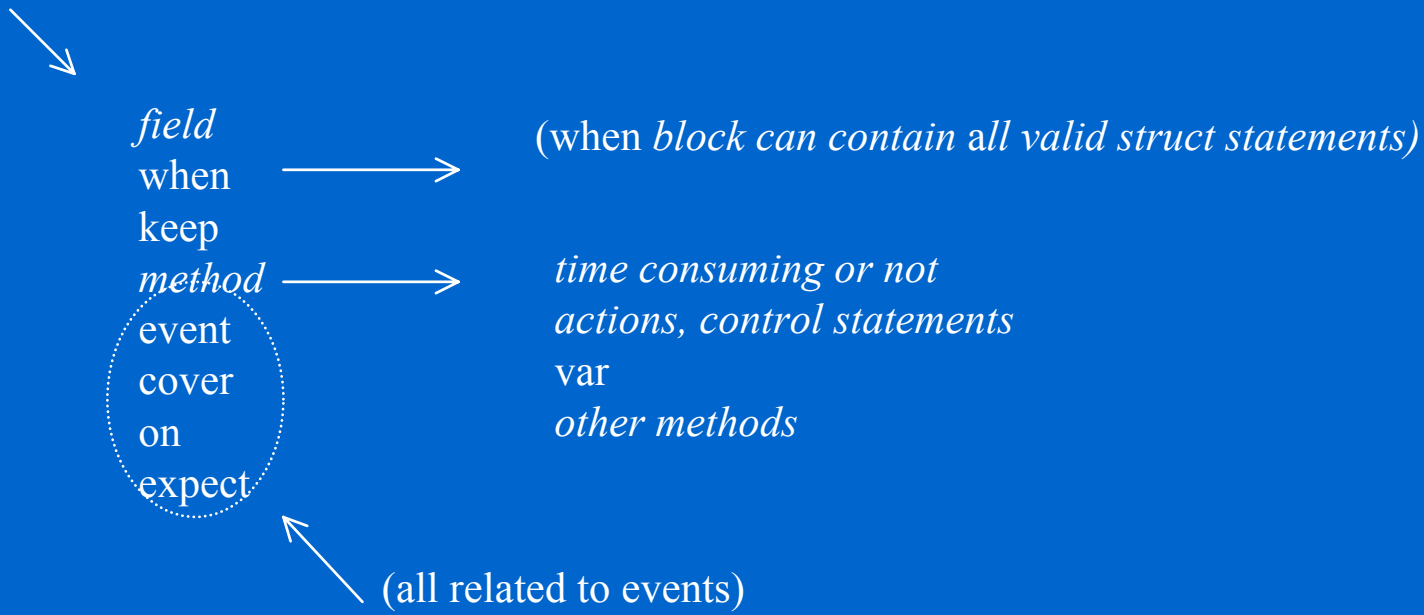
Intro to Specman Code

- This presentation is intended to be delivered by a Verisity CE to a customer who is going to start an evaluation.
  - It focuses on "what can be done" and not so much on syntax. Syntax can be found in the reference manuals.
  - It does not cover all that is available in Specman, just the most common constructs that will be needed to complete 90%+ of most environments. See reference manuals or CE for additional needs.
- This presentation may be used just to get an idea of the Specman constructs. For this purpose, it is recommended to see an example environment (such as the ATM environment) as well to get more examples and context.

Basic ‘e’ code structure

- At the ‘top-level’ of code, one can define a struct or use the type, define, import or verilog statements. Within a struct, there are a number of struct members that can be specified. Within a method, variables can be defined and action and control statements can appear, as well as other methods called.

struct type verilog import define



Minimal ‘e’ template for simulation

- This is the minimal template necessary to test a circuit with a VHDL or Verilog simulator. The next slides present topics in the order of this template

```
struct my_struct_s {
   field;  -- describe field to be passed in to simulation
   keep …; -- if any constraints on legal value of field
   event clk is rise('path/clk')@sim;
   method() @clk is {
      for or while loop {
         gen field;   -- only if generation-on-the-fly
         'path/input' = field; -- or = variable
         wait [1];    -- wait of at least one period needed
      };
      stop_run();
   };
   run() is also {start method() };
};
extend sys { inst_name : my_struct_s };
```

Fields Syntax

- Syntax
  - *field_name : type*;
    - type will determine width of field and how certain functions (print, for example) work with the value

Types

- Types
  - integer
    - int - 32-bit signed integer
    - uint - 32-bit unsigned integer
    - int (bits: *n)* - *n*-bit signed integer
    - uint (bits: *n)* - *n*-bit unsigned integer
  - integer subtypes
    - byte - 8-bit unsigned integer, same as uint (bits:8)
    - bit - 1-bit unsigned value, same as uint(bits:1)
  - Boolean
    - bool - 1 bit Boolean value (TRUE or FALSE)

Types

- Enumerated types
  - defined with type statement
    ```
    type color_t : [RED, GREEN, BLUE];
    ```
  - the enumerated type corresponds to a 32-bit value starting with 0 on the left and incrementing by 1 moving right
  - can modify the width when defining
    ```
    type color_t : [RED, GREEN, BLUE] (bits:8);
    ```
    or when using
    ```
    field_name : color_t (bits:8)
    ```
  - can specify other values
    ```
    type color_t : [RED=5, BLUE=23,GREEN=7];
    ```

Types

- structs
  - any user defined struct is also a valid type for a field

```
struct a_s {
   x : int;
};
struct b_s {
   y : a_s; -- the type is struct "a_s"
};
```

Lists

- List of any scalar type (integer, Boolean, enum) or of struct
  -
```
sizes : list of uint; -- list of unsigned integers
```
- List size can be specified in field definition
  -
```
colors [10] : list of color_t; -- list of 10 items of
type color_t
```

Field Examples

```
type size_t : [small,medium,large];
struct my_struct_s {
  a : int;                -- 32-bit signed integer
  b : uint (bits:4);      -- 4-bit unsigned integer
  c : byte;               -- 8-bit unsigned integer
  d : bool;               -- Boolean, TRUE or FALSE
  e : size_t;               -- enumerated type defined above
  f : list of byte;       -- list, unspecified depth, of byte
  g [8] : list of bit; -- list with 8 items, each 1-bit
};
struct other_struct_s {
  m : my_struct_s;          -- entire body of my_struct
};
```

Example Struct - Packet

- Create a struct called packet from the following specification:
  - Packet has three physical fields: a packet length (len), an address (where the data is routed to), and an array of bytes (data). Also specify a virtual field (kind) with an enumerated type to specify if it is a good or bad packet.
  - “len” should be 6 bits, “addr” should be 2 bits, “data” is of length “len”
- (How would one define a field in another struct to be a list called packets that contains 10 instances of packet?)

Packet Struct Solution

```
<'
type packet_kind_t : [GOOD, BAD];

struct packet_s {
   kind : packet_kind_t;
   addr : uint (bits : 2);
   len : uint (bits : 6);
   data [len] : list of byte;
};

struct my_stimulus_s {
   packets [10] : list of packet_s;
};
'>
```

Generation with basic structs

- We have just created a basic struct.

- If we generate “packets” now, each of the fields in packet will be generated with random data.
- We probably want to constrain the generation to only include legal values.

- We do that simply by extending our definition of packet through a construct called *keep*……

Constraints (keep) Overview

- Four styles of keep
  - `keep` *`Boolean-expression`*
    - constraint forces Boolean expression to be TRUE
  - `keep` *`Boolean-expression => Boolean-expression`*
    - if first Boolean expression is TRUE, constraint forces second Boolean expression to also be TRUE
  - `keep for each in` *`list_name`* `do`
    - apply constraints for each element in the list
  - `keep soft` *`field`* `== select { weight1 : val1; weight2 : val2 };`
    - change selection from flat distribution to weighted distribution
- All keeps must be satisfied, contradiction causes an error
  - `keep soft` is satisfied if possible, but discarded if contradicting

Keep Boolean expression

- ==, !=, >, <, >=,<=
  - `keep x == 25;`
    `keep 5 < x; -- order in expression not important`
    `keep y >= x + 9; -- arithmetic operations allowed`
    `keep y < my_method();`
    `keep z != large;`
- in range
  - `keep x in [5..10]; -- x is an int`
    `keep y in [1,3,5..8]; -- y is a uint`
    `keep z in [SMALL..MEDIUM]; -- z is enum`
  - `keep k not in [100..199]; -- k will not be 1xx`
- method that returns bool (see method section)
  - `keep my_list.is_all_iterations(.x);`

Keep implication

- Keep expression1 => expression2
  - expressions can have any of the forms on the previous page

```
keep x == 4 => y == 7;
keep x > 10 => y != x - 4;
```

Keep for each in list

- Keep for each (item_name) in list_name do { body; };
  - Can use Boolean expressions or implications from previous pages in body

```
keep for each (a) in alist do {a < 10};
keep for each (mystruct) in mylist do {
   index == 0 => mystruct.x < 40;
   index in [1..4] => mystruct.x == 10;
   index > 4 => mystruct.x != mystruct.y
};
```

  - Add "using index (*index_name*)" to use index value in body.

```
keep for each (a) using index(i) in alist do {a == i};
```

Keep - Weighted Distribution

- By default, each legal value has an equal chance of being selected. Sometimes, you want a different distribution.
  - ```
    keep soft port == select {
      30: inport0;
      50: inport1;
      10: inport2;
      10: others;
    }; -- port is enum with values inport0, inport1,...
    ```
  - Ranges can be specified
    - ```
      keep soft addr == select {
         8: [0..255];
         2: [256..4096];
      };
      ```

Constraints Example

- Constrain the packet struct created previously to:
  - Make the length less than 20
  - Generate about 90% good packets and also 20% address 0, 60% address1 and 20% address 2.

Constraints solution

```
<'
extend packet_s {
        keep len < 20;
        keep soft kind == select {
                90: GOOD;
                10: BAD;
        };
        keep soft addr == select {
                20: 0;
                60: 1;
                20: 2;
        };
};
'>
```

Simulator interface

- We have seen how we can easily control the generation of our stimulus.
- Now, how do we apply this generated stimulus to the simulator and the device under test?……..

Event “clock” definition

- You must define a Specman event tied to an HDL signal in order to drive or read HDL signals as simulator time advances.

```
event name is rise('path/clk')@sim;
```

  - choose the event name as you want
  - choose rise, fall, or change of the HDL signal (methodology recommendation: choose the opposite of the active edge inside the DUT)
  - the path should begin with ~ and use / as a hierarchy separator (e.g. ~/top/level1/level2)
  - the @sim is required (it causes the simulation callback when the signal change occurs)

Methods Overview

- We need a way to assign new values to Specman variables or simulator signals based on the values of other Specman variables or simulator signals -- methods.

- Methods can complete without simulator time advancing; we call these regular methods. Methods can also operate as simulator time advances, for example assigning a new value to a DUT input every clock cycle; we call these time-consuming methods or TCMs.

Specman Variables

- Methods can use/change the values of:
  - variables passed in as parameters
  - all fields in the struct in which the method is defined
  - fields in other structs specified with a full path name from sys
    - For example, if sys has a field a of type astruct, astruct has a field b of type bstruct, and bstruct has an integer field z, then a method in struct kstruct can access sys.a.b.z.
- You can also create variables of any type. Variables can be used/changed only in that method (local scope)

```
var name : type;
```

- For variables of type struct, use "= new"

```
var name : struct_name = new;
```

Simulator Signals

- Signals in the device-under-test can be accessed by describing the full Verilog/VHDL path and signal name, enclosed in single quotes. A style that works for Verilog and VHDL is to use "/" as the hierarchy separator and start the path with "~".
  - `'~/top/middle/bottom/my_signal'`
- You can also use the style for your simulator
  - e.g. for Verilog, `'top.middle.bottom.my_signal'`

Assignment

- Assignment (an action statement)
  - item = RHS
- item
  - field in this struct or with full path name from this struct or from sys
  - variable declared in this method
  - signal in the RTL ('~/full_path/signal_name')

NOTE: if item is a list or a struct, assignment will not create a copy of the list or struct; instead another pointer to the list or struct is created, meaning that a change to the underlying list or struct based on either pointer will be seen when the list or struct is viewed by either pointer. If you want a copy, see copy method described later.

Assignment (cont.)

- RHS
  - another item
  - any number of operators applied to fields or variables
  - a value returned by a user-defined or pre-defined method

Operations

- Operations for integers (includes uint, byte, bit)
  - +, -, *, / : addition, subtraction, multiplication, division
  - <<, >> : left shift, right shift
  - |, &, ^ : bitwise OR, AND, XOR
  - ~ : bitwise complement
- Operations for Booleans
  - and, or, not

Operations - example

- x = 2, y = 1, k = TRUE
  - ((x + y) * 2) - 5
    x << 3
    x | y
    (x == 3) or ((y ==1) and k)

List Elements and Bit Slicing

- List elements
  - Individual elements from a list can be specified by [*index*]
    - `my_list[5] -- 6th element in my_list (0 is 1st)`
  - Sublist can be specified by [*low..high*]
    - `my_list[2..5] -- 3rd through 6th element`
- Bit Slicing
  - A bit from an integer can be specified by [*bit_pos:bit_pos*]
    - `my_int[7:7] -- MSB of an 8-bit integer my_int`
  - A range of bits from an integer can be specified as [*high:low*]
    - `my_int[7:4] -- upper 4 bits of my_int`

Control Statements - Loops, Conditional

- Methods execute each line in their body and then quit
- Often, we want to execute lines more than once. This is where loops are useful.
    - for loop
    - while loop
- Sometimes we want to execute a group of lines only when a certain condition is true
    - if statement

For loops

- for *local_int* from *low* to *high* do { *action_block* };

```
for j from 0 to 10 do {
   print j;
};
```

- for each (*item_name*) in *list_name* do { *action_block* };

```
for each (databyte) in databytes do {
   sum = sum + databyte;
};
```

  - Add "using index (*index_name*)" to use index value in body.

```
for each (a) using index(i) in alist do {
  clist[i] = a | blist[i];
};
```

While loops

- while Boolean-expression do { block };

```
while x < 10 do { -- assuming x is 0, executes 10 times
   print x;
   x = x + 1;
};

while TRUE do { -- executes forever
   print sys.time;
   wait [1]; -- must be in time-consuming method
};
```

if..else conditional

- Sometimes, we want to perform the actions only under certain circumstances; use the “if...else” conditional
  - if Boolean-expression { block };

```
if a < 10 then {
   print a;
};
```

  - if Boolean-expression { block } else { block };

```
if a < 10 then {
   print a;
} else if a <= 15 then {
   out(“between 10 and 15”);
} else {
   out(“greater than 15”);
};
```

- (a case statement is also available)

Apply “packets” to simulator

- We have all of the constructs necessary to apply the packets created (from the generation and constraint examples) to the simulator… try it!
  - Send the 10 packets to a device that takes input once every rising edge of top/accept. The first period it wants to see the address on top/address and the length on top/length. Each subsequent period, it wants to see one byte of the data on top/databyte.
    - Hints:
      - Use the basic template.
      - Outer loop to iterate through packets, inner loop to iterate through each byte in data.

Simulator Interface Solution

```
struct stimulus_s {
   packets [10] : list of packet_s;
   event clk is fall('~/top/accept')@sim;
   drive() @clk is {
      for each (p) in packets do {
         '~/top/address' = p.addr;
         '~/top/length'  = p.len;
         wait [1];
         for each (d) in p.data do {
            '~/top/databyte' = d;
            wait [1];
         };
      };
      stop_run();
   };
   run() is also {start drive()};
};
extend sys { my_stimulus : stimulus_s };
```

Alternative: Using Verilog Tasks

- If you already have a Verilog task that implements the interface protocol, do you have to duplicate that in a method? -- No, just call the task!
  - Define the task

    ```
    verilog task 'taskname'(in_name:size,out_name:size:out);
    ```

    - all of the names and the sizes must match the task definition in the .v
    - in_name and out_name represent the parameters; you can have any number of parameters separated by comma; an output parameter must be flagged with ":out" as out_name is above
  - Call the task (as if it was a method) from a TCM

    ```
    'taskname'(addr,data);
    ```

Solution Using Verilog Task

```
verilog task 'top.send_pkt'(addr:2, len:6, data: 504);
struct stimulus2_s {
   packets [10] : list of packet_s;
   event clk is fall('~/top/accept')@sim;
   drive() @clk is {
      for each (p) in packets do {
         'top.send_pkt'(p.addr,p.len,p.data);
      };
      stop_run();
   };
   run() is also {start apply()};
};
extend sys { my_stimulus : stimulus2_s };
```

- NOTE: In Verilog, a task cannot have a parameter with variable width, so data is defined as its maximum width, 8 x (2**6 - 1)

Varying the Template

- You may want to modify the use of some of the other constructs in the template.
  - the wait command
  - the definition of the method
  - starting the method
  - extending structs and methods
- Here's a little more detail on these topics...

The Wait Action

- wait
  - valid only in time-consuming method (method definition defines default event)
  - wait for some number of cycles of the default event
    ```
    wait [5]; -- waits 5 cycles of default event
    ```
  - wait for another event
    ```
    wait until rise('~/top/rst'); -- wait for inactive
    wait until @interrupt;
    ```
  - wait for a condition to become true
    ```
    wait until true('~/top/flag' == 1);
    ```

Starting Methods

- Regular methods can be called by other regular methods or by time-consuming methods

```
x = f(a,b); -- this line could be in TCM or regular method
-- assume f is a regular method with a value returned
```

- Time-consuming methods can only be started (not called) by regular methods, although they can be either started or called by other time-consuming methods

```
start tcm1(a,b,c); -- valid in TCM or regular method
tcm1(a,b,c); -- valid only in TCM
-- assume tcm1 is a TCM with no value returned
```

  - NOTE: there is always an implied wait for the default event before the body of a TCM executes

Starting Methods in the First Place

- So, if each method is started/called by another method, how is the very first method started? The Specman test command starts the following built-in methods for the global struct and each struct in sys
  - init() - runs before pre-run generation
    - set up configuration parameters, assign initial values to fields (use ! so that generation does not overwrite these values)
  - post_generate() - runs after pre-run generation
    - modify values based on what generation has created
  - run() - runs just before simulation
    - start TCM's that should begin at simulation time 0

Extending Methods

- We need a way to add to the built-in struct methods. The way to extend any method (an exception: the list methods cannot be extended) is by using “is first” or “is also”
  - Use the same method definition as was used for the original definition except change “is” to “is first” to add to the beginning of the method or to “is also” to add the end of the method
  -
```
run() is also { start my_tcm() };
parity(data: list of byte) : byte is first {
   “some initialization code”
};
```

Extending Structs

- As with methods, we want to be able to extend structs.
  - sys is a built-in struct (already defined) so we can only extend it
  - user-defined structs might be extended in a different file than where they were defined to group code according to its purpose rather than its struct
    - `struct my_struct_s { body }; -- defined somewhere`
      `extend my_struct_s { any_struct_member };`
    - ```
      extend sys {
         my_inst : my_struct_s;
      }; -- puts an instance of my_struct in sys
      ```
      - now my_inst fields will be generated and its built-in methods run
      ```

Make Your Own Methods

- So far, we have focussed on what can go in the body of a method. What about the definition of the method?
  - Required
    - name of the method
    - parameters and their types (if any)
    - return value type (if any)
    - if and only if this a TCM, the default event
  - *method*(*param1* : *type*, *param2* : *type*) : *type* *@event* is { *body* };

```
parity(data: list of byte) : byte is {…
apply() @sys.clock is {…
```

  - If a return value type is defined, a variable "result" is implicitly declared

```
result = param1 + param2;
```

Method example - packet

- A common need for a user-defined method is to calculate a constraint value using an algorithm that is too complex to state in a single line.
- Create a method in the sys struct to calculate parity for the data field of packet.
  - The parity algorithm is a simple bitwise XOR of the bytes in data, yielding a byte. Make a field called parity in the packet struct and constrain its value to equal the return from the method when the kind field has the value "good".
  - Hint: think of the
    - input parameter and its type; the return type
    - way to iterate through each value in a list

Parity solution

```
extend sys {
   parity_calc (data : list of byte) : byte is {
      for each (item) in data do {
         result = result ^ item;
      };
   };
};

extend packet_s {
   parity : byte;
   keep kind == good => parity == parity_calc(data);
};
```

Generation On-The-Fly

- Generation on-the-fly means creating stimulus just before it will be applied.
  - Advantages:
    - can base generation results on current state of simulation
    - saves memory versus generation all stimulus before the run
  - Use the “gen” action
    - get the generator to produce a new value for a field (using all of the existing constraints) or for a variable
    - can specify new constraints with gen keeping

```
gen x keeping {it in [1..5]};
gen mystruct keeping {.a < 50; .b > 4};
```

Generation On-The-Fly Example

- Try rewriting the simulator interface as generation on-the-fly.
    - Send 15 packets. For each packet, set the addr value to the value of the HDL signal “top/two_bits”.

Interface Solution Using Gen On-The-Fly

```
struct stimulus3_s {
   event clk is fall('~/top/accept')@sim;
   drive() @clk is {
      var next_pkt : packet_s;
      for i from 1 to 15 do {
         gen next_pkt keeping {.addr == '~/top/two_bits'};
         '~/top/address' = next_pkt.addr;
         '~/top/length'  = next_pkt.len;
         wait [1];
         for each (d) in next_pkt.data do {
            '~/top/databyte' = d;
            wait [1] * cycle;
         };
      };
      stop_run();
   };
   run() is also {start apply()};
};
extend sys { my_stimulus : stimulus3_s };
```

Concatenating with pack

- To you, there are separate fields, to the device it is a single stream of bits; how can you make the translation? The built-in method called pack.
  - Pack will take fields or variables and concatenate them into an integer, a list of bits, or a list of bytes as needed. There are various styles of packing (big/little endian, etc.).
  - To save typing, you can specify an entire struct to be packed, and all the fields marked with "%" will be packed
- `pack(packing_style, item1, item2,…);`

Packing Styles

- packing.high
  - the first field goes in the most significant location of the result, the second field goes in the next most significant location, and so on until the last field goes in the least significant location
  - use for concatenating items to an integer
- packing.low
  - the first field goes in the least significant location of the result, the second field goes in the next least significant location, and so on until the last field goes in the most significant location
- make your own
  - your CE can help you create a new style in a setup file

Pack Examples

- a = 10101010; f = 1111; c is list of bit, c[0]=1, c[1]=1, c[2]=0,c[3]=0; x is list of byte
  - x = pack(packing.high, a, f, c)
    - x[1] = 10101010, x[0] = 11111100
  - x = pack(packing.low, a, f, c)
    - x[1] = 00111111, x[0] = 10101010
- Modify the simulation interface solution assuming the device now wants to see address and length concatenated as the first data byte
    - Hint: use pack to concatenate address, length and data into a single list of byte, then apply each byte in a loop

Interface Solution Using Pack

```
struct stimulus4_s {
   packets [10] : list of packet_s;
   event clk is fall('~/top/accept')@sim;
   drive() @clk is {
      var bytes : list of byte;
      for each (p) in packets do {
         bytes = pack(packing.low, p.len, p.addr, p.data);
         for each (b) in bytes do {
            '~/top/databyte' = b;
            wait [1];
         };
      };
      stop_run();
   };
   run() is also {start apply()};
};
extend sys { my_stimulus : stimulus4_s };
```

Pre-defined List Methods

- With “for each..in list”, you can do all kinds of processing on a list; however, common list operations have already been implemented in built-in methods for you
  - every list you create has the following methods available to it (see next page, ref. manual for more)
    - copy()
      - return a new list with the same element values as the original
    - size()
      - return the number of elements in the list
    - add(element)
      - add the element to the end of the list

More List Methods

- more list methods
  - all/ has/ first(expression)
    - evaluate the expression for each of the elements in the list and return a sublist of all elements for which the expression is true (all), a Boolean saying whether any element has that expression true (has), or the element itself which is the first one that has the expression true (first)
  - delete(index)
    - delete the element with that index
  - pop/clear()
    - return the last element in the list and remove it from the list (pop); remove all elements from the list (clear)
  - crc_8, crc_32
    - compute common CRC equations over a list of bits (see ref. for syntax)

Printing actions

- For debugging and for generally following the progress of the simulation, we want to get text output from Specman. There are various print actions available.
  - print
    - use the Specman print function on a field or variable
    - output style for structs can be modified
  - out
    - simple printing of a simple values (a struct or list will be printed as a pointer; carriage return is automatically added at the end
  - outf
    - like out, but c-type formatting is allowed and there is no automatic carriage return at the end

Printing examples

- print
  - `print index using dec;`
    `print data using hex;`
- out
  - `out("Index is" , index, "and data is ", data[index]);`
- outf
  - `outf("Index is %d and data is %x\n", index,data[index]);`

Checking action

- To verify that all values are as expected, use the check action
  - `check` *`Boolean-expression`* `else dut_error("`*`message`*`");`
- Boolean expression is no different from elsewhere in code
  - Boolean variable, Boolean variables combined with Boolean operators, or `method` returning Boolean
  - Two or more non-Boolean values with compare operator
  - Common use is e-value == 'simulator value'

```
check that reg == '~/top/reg'
   else dut_error("reg mismatch");
```

- Message is same syntax as for out() action