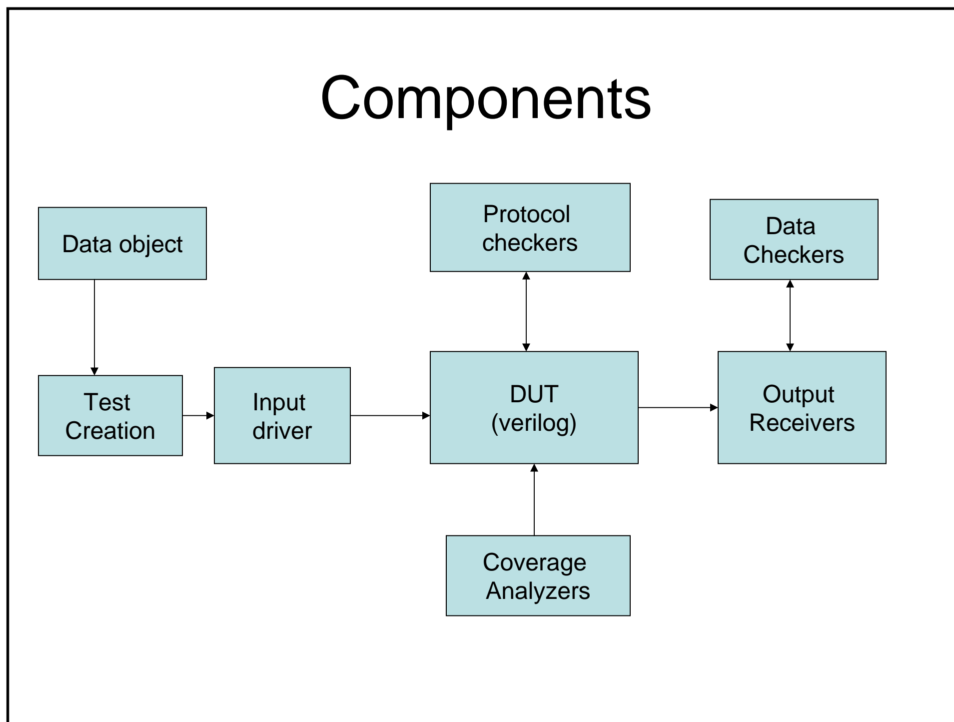
Design
Verification with ‘e’

The language ‘e’

- Contains *all* the constructs necessary for a complete verification tool
  - Allows objects in the verification environment to be extended
  - Needs to express constraints
  - Coverage engine
  - Temporal engine : Capture protocols rules in a concise fashion
- Specman elite tool from Verisity Design supports the e language



Data Object

- Used to represent stimulus item or one test vector
- Different tests have types of values for this field
  - Router: packets
  - Video processor: frames
  - Micro-processor: instructions
- Random stimulus item generator

Test creation

- Set of constraints placed on the generation of the fields in the data object
- More restrictive constraints lead to directed testing

Unit vs Struct

- Unit is very similar to struct
- Static verification object that does not move through the verification system
- A struct is a dynamic object such as a packet or instruction that moves through the system

Input Driver

- A *unit* is defined to represent the driver
- Input procedure which takes one stimulus item and applies to the DUT
- Also has a procedure to call the input procedure multiple times to apply many stimulus item to the DUT

Output Receiver

- A unit to represent a receiver object
- Procedure to collect raw output from DUT and convert to a data object format
- Has to follow the interface protocol expected by the DUT at the output port
- Receiver then passes this data object to the data checker to compare against expected data

Data checker

- A unit to represent a data checker object
- Gets an output data from the receiver and compares with the expected data
- Has a procedure to generate and store the expected data
- May be instantiated in the receiver object
- Or a centralized object instantiated directly

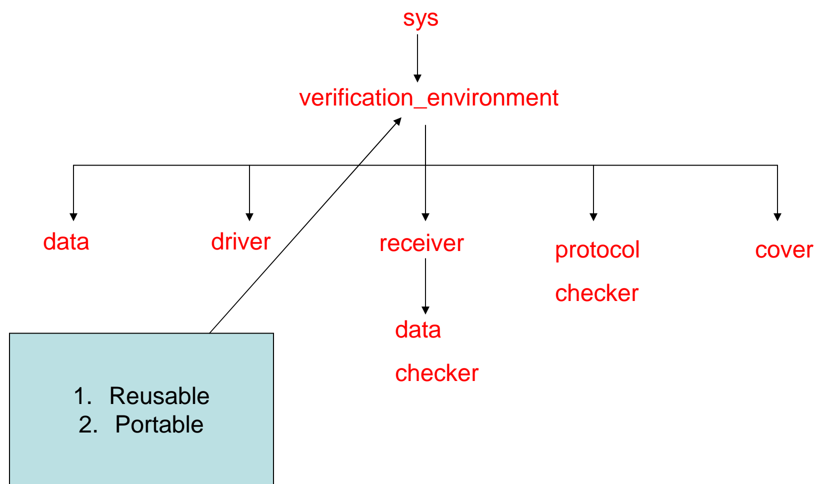
Protocol Checker

- A unit is used
- Monitors that the protocols at the input and output interfaces are not violated

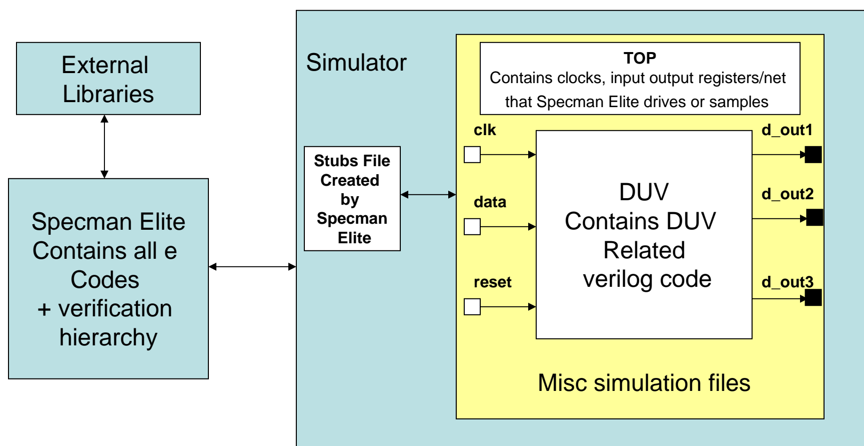
Coverage Analyzer

- Defines a set of basic, transition and cross-coverage items to monitor specific events in the simulation
- Statistics displayed after simulation for further testing

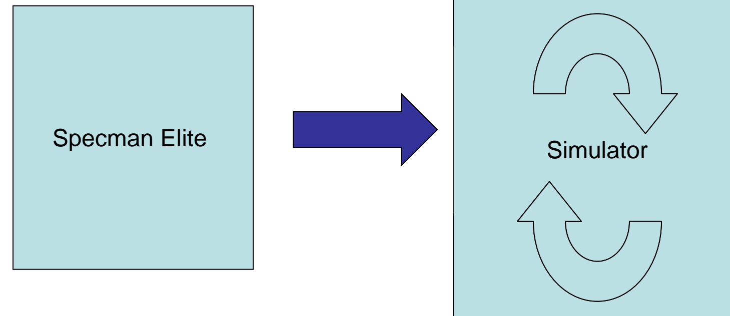
e Hierarchy



Interaction Between Specman Elite and Simulator



Flow of Simulation



Syntax of ‘e’

Naming Conventions

- User defined names, struct names : all start with CAPITAL letters
- User defined struct member names (fields, events, methods): all start with LOWER case letters
- In both cases each of the parts of the name start with CAPITAL letters
- HDL signals : all lower case, parts of the name are separated with underscores

Naming example

```
<'

 // type names start with a capital letter each part of a name
starts with capital letter...therefore PacketSize and not
Packetsize, Packet_size or anything else
  type PacketSize : [small, medium, big];

  // unit and struct names start with a capital letter
 unit ChipEnv {

  // fields, events, methods (in one word - struct members)
 // start with lower case letter
    chipAddress : uint;
    event configDone;
```

Naming Example

```
resetChip() is {
    // a reference to an HDL signal name,
    // all lower case, parts separated by underscores
      'sys_reset' = 0;
    };
// another user defined method
endSim() is {
•        // a method that is a part of the e language - follows
•        // the e proposed standard naming conventions
•        // (names are all lower case,
•        // parts of a name are separated by underscores)
dut_error("simulation ended without error");
   };
};

•   '>
```

Directed vs Random Verification

- Directed: The designer has to think about the places where the bug can hide and then write test-benches to check the design at those points
- But a DUV may behave differently if a different set of inputs are applied to reach those points
- Writing requires knowledge of the design
- Also with the increase in the number of test cases for a million transistor design directed test benches require lot of time

Hence Random

- Constrained Random Testbenches:
  - Provide constraints or limits to the inputs
  - Within this limit generate inputs randomly
- May want dependencies…
- Takes care of Simulation Time
- "Garbage Collection": no memory leak

AO vs OO

- AO: One can 'extend' data structures and methods in another file
- Example:

```
<' struct Fruits{
     apples : uint;
     oranges : uint;
     sumFruitsUp() : return uint is{
       result = apples + oranges;
       };
   };
'>
```

AO coding

```
<'extend Fruits{
    mangoes : uint;
    papayas :  uint;
    sumFruitsUp(): return uint is also {
        result += mangoes + papayas;
    };
};
>'
```

Packet.e : The OO approach

```
<'
struct Packet {
        %header : byte;
        %payload : list of byte;
         keep payload.size() < 200;
         %crc : byte;
  //this method describes packet conversion
  convert () : Packet is{ };
  //this method converts o/p data from DUV to packet structure
  rebuild(packetBytes : list of byte) : Packet is{
   };
};
'>
```

Packet.e : AO approach

```
<‘
   struct Packet{
   %header : byte;
   %payload : list of byte;
   keep payload.size() < 200;
   %crc : byte;
};
‘>
```

DataChecker.e

```
<‘
    extend Packet{
       convert() : Packet is {
        };
     rebuild(packetBytes: list of byte): Packet is{
    };
};
‘>
```

Data Types

- Scalar Types
  - Numeric
    - length: int; //int -> numeric data, size = 32 bits
    - addr:uint(bits:24); //unsigned + number, size=32bits
    - valid:bit; //1 bit field
  - Boolean=> frame_valid: bool;//logical value
  - Enumerated
- List Types
- String Types

Enumerated Data Types

- <‘
  type Packet_protocol: [ ];
  ‘>
- <‘
  extend Packet_protocol:[Eth, IEEE];
  struct Packet{
  kind : Packet_Protocol;
  };
  ‘>

Enumerated Date Type

- Defines the valid values for variable or field as a list of symbolic constants
- ```
  <'
  type Instr_kind: [imm, reg];
  '>
  ```
- ```
  <'
   type Instr_kind: [imm=4, reg = 8];
   '>
  ```

Scalar Subtypes

Because and & or are 'e' inbuilt

```
<'
type Opcode: [add, sub, or1, and1];
type Logical_opcode: Opcode [or1, and1];
type Small: uint(bits:4);
struct instruct{
op1: Opcode;
op2: Logical_opcode;
length: Small;
}
'>
```

List Types

- Hold ordered collection of data elements

```
<'
 struct Packet{
  addr : uint(bits:8);
  data1: list of byte;
  };
struct Sys{
packets[10]: list of Packet;
values: list of uint(bits: 128);
};
'>
```

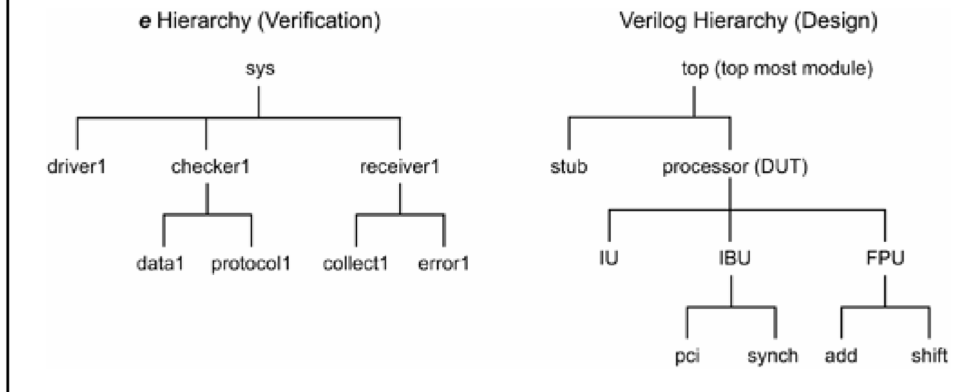
String Type

```
<'
 struct Dtypes{
    m() is{ //METHOD
    var message: string;
    message="This is string";
    print message;
    };
 };
'>
```

NOTE: var is used to declare a variable inside a method

Verification with e

- e & HDL Hierarchy
  - "sys" is implicitly defined
  - topmost struct in Verification environment



Verification with e

```
<'

struct Data {
   <data internals>
};

struct Protocol {
   <protocol internals>
};

struct Collect {
   <collect internals>
};

struct Error {
   <error internals>
};

struct Driver {
   <driver internals>
};
```

Verification with e

```
struct Checker {
   data1: Data;
   protocol1: Protocol;
   <checker internals>
};

struct Receiver {
   collect1: Collect;
   error1: Error;
   <receiver internals>
};
```

```
extend sys {
   driver1: Driver;
   checker1: Checker;
   receiver1:Receiver;
};

'>
```

Verification with e

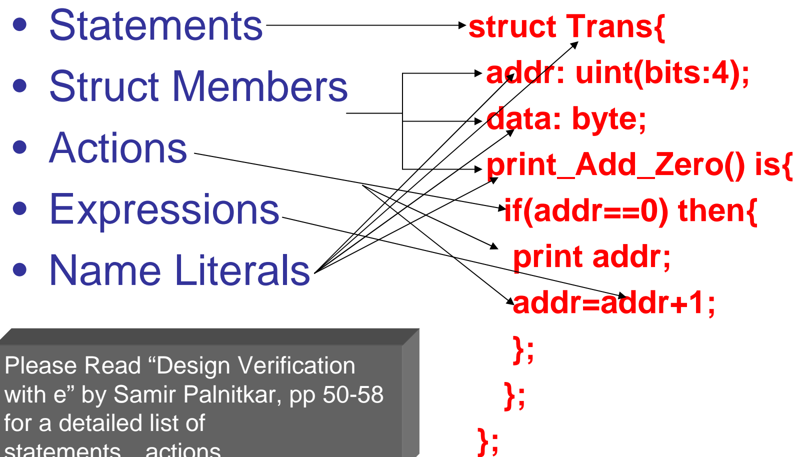
- e lends itself very well to a layered development approach
  - The functionality for the base structs can be extracted from the design specification
  - Test-specific changes can then be added separately using the extend mechanism for the structs.

Driving & Sampling DUT Signals

```
<‘
struct Driver{//Struct in the e environment
 r_Value : uint(bits:4);
read_Value() is{
r_Value=‘~/top/processor/FPU/add/operand’;
};
write_Value() is{
‘~/top/processor/FPU/add/operand’=7;
};
‘>
```

Computed Signal Names

```
<‘
struct Driver{
 id: uint(bits:2);
 r_Value: uint(bits:4);
 read_Value() is{
   r_Value=‘~/top/processor_(id)/FPU/add/operand’;
 };
‘>
```



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
- Associated with a method, event or an "on" struct member

Example

```
<' struct Packet{
       event xmit_Ready is rise('~/top/ready');
       length: byte;
       delay: uint;
       on xmit_Ready{
          transmit();
       };
       transmit() is{
          length=5;
          delay=10;
          out("transmitting packet");
         };
};
'>
```

Types of Actions

- Creating or modifying variables
- Executing actions conditionally
- Executing actions iteratively
- Invoking methods and routines: invokes methods and routines
- Time Consuming Actions: causes simulation time to elapse before a callback is issued by the simulator
- Generating Data Items
- General Options

Time Consuming Actions

- emit: causes a specified event to occur
- wait: suspends execution of the current TCM until a given temporal expression succeeds
- all of: executes multiple action blocks concurrently, as branches of a fork. Following action is reached only when all the branches have executed
- first of: Following action is reached when any of the branches of the first of block have been executed

Highlights

Example of using events in Time Consuming Methods (TCM)

```
<'
struct meth {
   event pclk is rise('~/top/pclk')@sim;
   event ready is rise('~/top/ready')@sim;
   event init_complete;
   my_tcm() @pclk is {
      wait @ready;
      wait [2];
      wait [3] @ready;     //Wait for three occurrences of @ready
      init_dut();                          //Call a regular method
      emit init_complete; //Manually trigger the event
   };
};
'>
```

Commenting in ‘e’

```
<'
       -- Single line comment
        // This is also single line comment
        //Code here
'>
Multipe line comment So ignored
<‘
 //Code continue..
 '>
```

‘e’ Operators

```
<'
 extend sys {
  run() is also {
  var a : byte;
  var b : byte;
  var c : byte;
  a = 0xAA;
  b = 0x55;
  outf (" a = %b b = %b\n", a, b);
  // Bitwise negation
  c = ~a;
   outf ("Bitwise negation a  is     :%b\n", c);

  // Bitwise AND operation
```

Bitwise Operators

```
  c = a & b;
  outf ("Bitwise AND of a with b is :%b\n", c);
  // Bitwise OR operation
  c = a | b;
  outf ("Bitwise OR of a with b is  :%b\n", c);
  // Bitwise XOR operation
  c = a ^ b;
  outf ("Bitwise XOR of a with b is :%b\n", c);
  // Left shift
   c = a << 2;
   outf ("Left shift a by 2 bits is  :%b\n", c);
    // Right shift
     c = b >> 2;
     outf ("Right shift b by 2 bits is :%b\n", c);
   };
};
'>
```

Boolean Operators

```
<'
extend sys {
 run() is also {
   var a : bool;
   var b : bool;
   var c : bool;
   var d : bool;
   outf (" a = %b b = %b c = %b\n", a, b, c);
   // Not Operator
   d = !(TRUE);
   outf ("Not of  (TRUE) is     :%b\n", d);
   // Boolean AND operation
   d = a && c;
   outf ("Boolean AND of a with c is :%b\n", d);
```

Boolean operators

```
// Boolean OR operation
  d = a || b;
  outf ("Boolean OR of a with b is  :%b\n", d);
  // Boolean implication operation
  d = (2 > 3) => (3 > 2);
  outf ("Boolean implication of (2 > 3) => (3 > 2) :%b\n", d);
  // Boolean implication operation
  d = (4 > 3) => (3 > 4);
  outf ("Boolean implication of (4 > 3) => (3 > 4) :%b\n", d);

 };
};
'>
```

'=>' : Returns TRUE when the first expression of two expressions is FALSE, or when both expressions are TRUE.

```
<'
type packet_protocol : [ETHERNET, IEEE, ATM];

struct Packet {
  protocol     :  packet_protocol;
  // payload is list of bytes
  // Which size is always 10 bytes
  payload      : list of byte;
  keep payload.size() == 10;
};
// Just to check our code
extend sys {
  // Create the list of the packets
  data : list of Packet;
  // Set number of packets to generate to 4
  // i.e. set the size of list
  keep data.size() == 4;
  run() is also {
    gen data;
    for each in data do {
      print it;
      print it.payload;
    };
  };
};
'>
```

An Example on Lists

```
Generating the test using seed 1...
Starting the test ...
Running the test ...
 it = packet-@0: packet
    ----------------------------------------------         @new
0    protocol:                  IEEE
1    payload:                   (10 items)
 it.payload =  (10 items, dec):
             133 185  157 142 231 104   85 230 102 168   .0

 it = packet-@1: packet
    ----------------------------------------------         @new
0    protocol:                  ATM
1    payload:                   (10 items)
 it.payload =  (10 items, dec):
              2  44  224 216 156  14  216  12  52  80       .0

 it = packet-@2: packet
    ----------------------------------------------         @new
0    protocol:                  Ethernet
1    payload:                   (10 items)
 it.payload =  (10 items, dec):
             112 201  150  25 244 227  194 171  77  96     .0

 it = packet-@3: packet
    ----------------------------------------------         @new
0    protocol:                  Ethernet
1    payload:                   (10 items)
 it.payload =  (10 items, dec):
              69 214   58 191 194 192   64 252 143  82     .0
```

Creating Hierarchy with Structs and Units

Structs

- Used to define Data Elements and the behavior of components in a verification environment
- Syntax:
  - struct struct-type {

    struct-member1;

    struct-member2;

    ….};

Struct Members

- Data Fields for storing data
- Methods for procedures
- Events for defining temporal triggers
- Coverage groups for defining coverage points
- when, for specifying inheritance subtypes
- declarative constraints for describing relations between data fields

Struct Members

- on, for specifying actions to perform upon event occurences
- can be empty also, for future extensions

An example

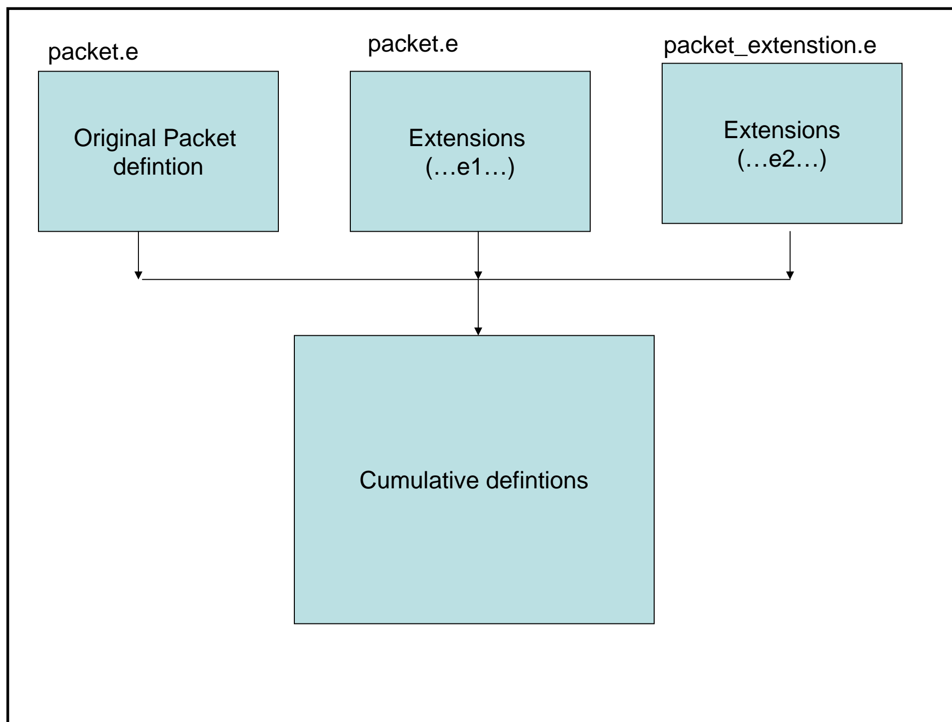
```
<'
 type Packet_kind:[ATM, ETH];//enumerated type
struct Packet{
  len : int; //field of struct
  keep len < 256; //constraint on struct member
  kind: Packet_kind; //field of struct
  calc_par() is{ //method (procedure) in a struct
  }; //end of method definition
}; //end of packet struct
'>
```

Extending Structs

```
//extend in the same file
extend sys{ //extend sys structure, which is empty initially
    packets: list of Packet;
    keep packets.size()==10;
    run() is also{
       print packets;
      };
};
extend Packet{
   addr : byte; // add a byte of address
   keep len < 128; // extend original packet definition
};
'>
```

Extend (in a different file)

```
packet_extenstion.e
<'   import packet.e; //import the original
                       //packet defintion
  extend Packet{
     keep kind == ATM;  //add constraints
     keep len == 64;  //to the packet definition
   };
'>
```



Fields inside a struct

- ‘!’ indicates Ungenerated field
  - Not assigned any value during the generated phase
  - Useful for fields assigned during the simulation
  - Default value is 0 (NULL)
  - For values specified in range, it’s the first value
    - ! num : int [10..15](bits:4);

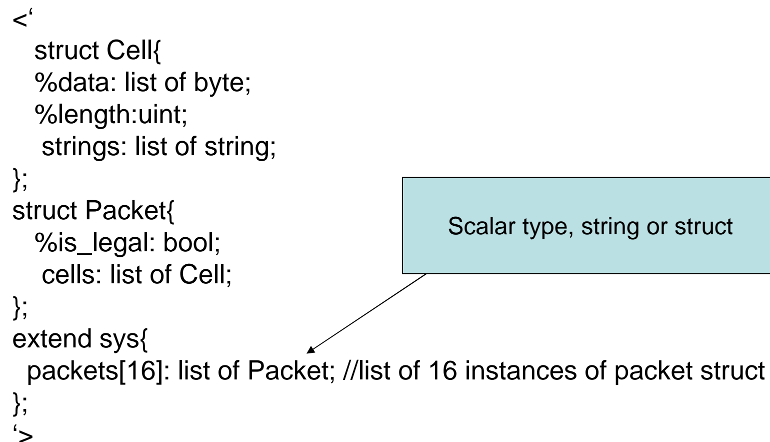
Fields inside a struct

- ‘%’ indicates Denotes a physical field. These fields, as opposed to virtual fields are sent to the DUT.
- Order of % and ! is immaterial
- field-name
- type
- min-val..max-val
- (bits|bytes:num)

Example of field definitions

```
<' type NetworkType: [IP=0x0800,ARP=0x8060](bits:16);
struct Header{
   %address: uint (bits:48);   // Physical Fields
   %length:uint[0..32];
};
struct Packet{
  hdr_type: NetworkType;   //Virtual field
  %hdr: Header; //Physical Field
  is_legal:bool; //Boolean Virtual Field
  !counter:uint; //Not generated when the packet instance is generated
};
extend sys{
  packet_i : Packet; //As packet is a struct such a field definition is called
                              //"instantiation"
};
'>
```

List Fields



List Operations

- size() : This is used to set the size of the list.
- add(item or list) : Add an item to the end of a list
- add0(item or list) : Add an item to the head of a list
- clear(): Delete all items from a list
- delete(index) : Delete an item from a list
- insert(index,item) : Insert an item in a list at a specified index
- pop(item) : Remove and return the last list item
- pop0(item) : Remove and return the first list item
- push(item) : Add an item to the end of a list
- push0(item) : Add an item to the head of a list
- resize() : Change the size of a list

AN EXAMPLE WITH LIST *METHODS*

```
<'
 struct packet{
   a_list:list of int;
   keep a_list.size()==3; //need to use "keep" outside a method
   keep a_list=={10;20;40};
   list_method() is
     {
       out("Printing List_method\n");
       var i_list:list of int(bits:5);
       i_list={2;4;6;8};
       var a:int;  // var is used inside a method, dont use "keep" here
       a = 2;      //These variables are like local variales
       var i:int;
       print a;
       print i_list;
       i_list.delete(2);
       print i_list;
       i_list.add(1);
       print i_list;
       i_list.push(1);
       print i_list;
       i_list.push0(1);
       print i_list;
       a=i_list.pop();
       print i_list;
```

```
     print a;
       var cnt:int;
       cnt = i_list.size();
       print cnt;
       out("\n Binary \n");
       for {i=0;i<cnt;i=i+1}
        do {outf("%b \t",i_list[i]);};
       out("\n");
       gen i_list keeping {it.size()==10;};
       print i_list;
     };
  };


  extend sys{
   data: list of packet;
   keep data.size() == 4;//there will be 4 packets generated
   run() is also{
    gen data;
    for each in data do{
      print it;
      print it.a_list;
      it.list_method(); //invoke the method, cant access the variables

    };
   };
  };

'>
```

```
Welcome to Specman Elite (4.3.6)  -  Linked on Mon May 30 11:24:07 2005

Loading listfunctions.e ...
read...parse...update...patch...h code...code...clean...GC(sys)...

Doing setup ...
Generating the test using seed 1...
Starting the test ...
Running the test ...
  it = packet-@0: packet
        ----------------------------------------------        @listfunctions
0       a_list:                    (3 items)
  it.a_list =
0.      10
1.      20
2.      40
Printing List_method

  a = 2
  i_list =  (4 items, dec):
                                  8  6  4  2          .0

  i_list =  (3 items, dec):
                                     8  4  2          .0

  i_list =  (4 items, dec):
                                  1  8  4  2          .0

  i_list =  (5 items, dec):
                               1  1  8  4  2          .0

  i_list =  (6 items, dec):
                            1  1  8  4  2  1          .0

  i_list =  (5 items, dec):
                               1  8  4  2  1          .0
```

```
a = 1
  cnt = 5

 Binary

1       10          100              1000            1

  i_list =  (10 items, dec):
              -10  15   4  -7  -1   9   9   5 -11   8      .0

  it = packet-@1: packet
        ----------------------------------------------        @listfunctions
0       a_list:                    (3 items)
  it.a_list =
0.      10
1.      20
2.      40
Printing List_method

  a = 2
  i_list =  (4 items, dec):
                                  8  6  4  2          .0

  i_list =  (3 items, dec):
                                     8  4  2          .0

  i_list =  (4 items, dec):
                                  1  8  4  2          .0

  i_list =  (5 items, dec):
                               1  1  8  4  2          .0

  i_list =  (6 items, dec):
                            1  1  8  4  2  1          .0

  i_list =  (5 items, dec):
                               1  8  4  2  1          .0
```

```
  a = 1
  cnt = 5

 Binary

1       10          100              1000            1

  i_list =  (10 items, dec):
                6   4   7  11 -15  -8   2   3 -15 -13           .0

  it = packet-@2: packet
        ----------------------------------------------        @listfunctions
0       a_list:                    (3 items)
  it.a_list =
0.      10
1.      20
2.      40
Printing List_method

  a = 2
  i_list =  (4 items, dec):
                                  8  6  4  2          .0

  i_list =  (3 items, dec):
                                     8  4  2          .0

  i_list =  (4 items, dec):
                                  1  8  4  2          .0

  i_list =  (5 items, dec):
                               1  1  8  4  2          .0

  i_list =  (6 items, dec):
                            1  1  8  4  2  1          .0

Checking is complete - 0 DUT errors, 0 DUT warnings.
```

```
 i_list =  (5 items, dec):
                               1  8  4  2  1          .0

  a = 1
  cnt = 5

 Binary

1       10          100              1000            1

  i_list =  (10 items, dec):
              4   2   6  -9 -12  -8  -12   6  12  -6      .0

  it = packet-@3: packet
        ----------------------------------------------        @listfunctions
0       a_list:                    (3 items)
  it.a_list =
0.      10
1.      20
2.      40
Printing List_method

  a = 2
  i_list =  (4 items, dec):
                                  8  6  4  2          .0

  i_list =  (3 items, dec):
                                     8  4  2          .0

  i_list =  (4 items, dec):
                                  1  8  4  2          .0

  i_list =  (5 items, dec):
                               1  1  8  4  2          .0

  i_list =  (6 items, dec):
                            1  1  8  4  2  1          .0

  i_list =  (5 items, dec):
                               1  8  4  2  1          .0
```

```
a = 1
  cnt = 5

 Binary

1        10            100                   1000                 1

  i_list =  (10 items, dec):
                      4 -15   9  -9 -14  -6  -14  -7  -6   9              .0

No actual running requested.
Checking the test ...
Checking is complete - 0 DUT errors, 0 DUT warnings.
Doing setup ...
Generating the test using seed 1637073683...
Starting the test ...
Running the test ...
  it = packet-@4: packet
          ---------------------------------------------                @listfunctions
0        a_list:                    (3 items)
  it.a_list =
0.       10
1.       20
2.       40
Printing List_method

  a = 2
  i_list =  (4 items, dec):
                                       8   6   4   2                    .0

  i_list =  (3 items, dec):
                                           8   4   2                    .0

  i_list =  (4 items, dec):
                                       1   8   4   2                    .0

  i_list =  (5 items, dec):
                                   1    1   8   4   2                   .0

  i_list =  (6 items, dec):
                               1   1    8   4   2   1                   .0

  i_list =  (5 items, dec):
                                   1    8   4   2   1                   .0
```

```
  a = 1
  cnt = 5

 Binary

1        10            100                   1000                 1

  i_list =  (10 items, dec):
                      -11 -11   -1   5   4  10    3  13  -5  -5        .0

  it = packet-@5: packet
          ---------------------------------------------                @listfunctions
0        a_list:                    (3 items)
  it.a_list =
0.       10
1.       20
2.       40
Printing List_method

  a = 2
  i_list =  (4 items, dec):
                                       8   6   4   2                    .0

  i_list =  (3 items, dec):
                                           8   4   2                    .0

  i_list =  (4 items, dec):
                                       1   8   4   2                    .0

  i_list =  (5 items, dec):
                                   1    1   8   4   2                   .0

  i_list =  (6 items, dec):
                               1   1    8   4   2   1                   .0

  i_list =  (5 items, dec):
                                   1    8   4   2   1                   .0

  a = 1
  cnt = 5
```

```
Binary

1        10            100                   1000                 1

  i_list =  (10 items, dec):
                      13  12    0 -11   4 -15    0   2   9  -1        .0

  it = packet-@6: packet
          ---------------------------------------------                @listfunctions
0        a_list:                    (3 items)
  it.a_list =
0.       10
1.       20
2.       40
Printing List_method

  a = 2
  i_list =  (4 items, dec):
                                       8   6   4   2                    .0

  i_list =  (3 items, dec):
                                           8   4   2                    .0

  i_list =  (4 items, dec):
                                       1   8   4   2                    .0

  i_list =  (5 items, dec):
                                   1    1   8   4   2                   .0

  i_list =  (6 items, dec):
                               1   1    8   4   2   1                   .0

  i_list =  (5 items, dec):
                                   1    8   4   2   1                   .0
 a = 1
  cnt = 5

 Binary

1        10            100                   1000                 1
```

```
  i_list =  (10 items, dec):
                      -2 -13   -5 -14  -6   0  -13 -11  -1  -3        .0

  it = packet-@7: packet
          ---------------------------------------------                @listfunctions
0        a_list:                    (3 items)
  it.a_list =
0.       10
1.       20
2.       40
Printing List_method

  a = 2
  i_list =  (4 items, dec):
                                       8   6   4   2                    .0

  i_list =  (3 items, dec):
                                           8   4   2                    .0

  i_list =  (4 items, dec):
                                       1   8   4   2                    .0

  i_list =  (5 items, dec):
                                   1    1   8   4   2                   .0

  i_list =  (6 items, dec):
                               1   1    8   4   2   1                   .0

  i_list =  (5 items, dec):
                                   1    8   4   2   1                   .0

  a = 1
  cnt = 5

 Binary

1        10            100                   1000                 1

  i_list =  (10 items, dec):
                      -9  12    1   8 -14   9    0  15 -16  -8        .0

No actual running requested.
Checking the test ...
```

Examples on list methods

- var i_list:list of int;
  i_list.add(5); //add the item or list to the
  //end; has to be list of same type
- var i_list:list of int;
  i_list.add0(5); //add to the start of the list
- var i_list:list of int;
  a_list.clear(); // deletes all items from list

Keyed Lists

- Enable faster searching of lists by designating a field or value to be searched
- Eg: sparse memory implementation
  – List which has a large capacity. Say is has small amount of data spread across.
- Syntax:

![%]list-name: list(key:key-field) of type;

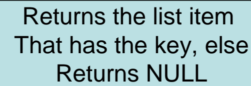
Struct members for structs; it for scalars

Cannot generate this field

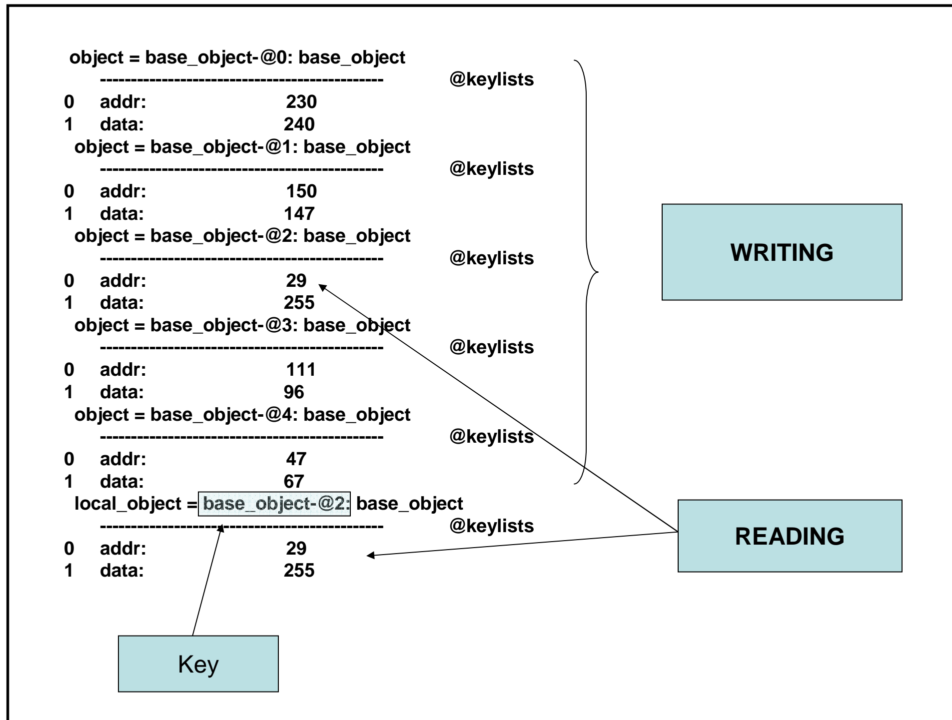
Scalar type, string or struct

Keyed Lists are used similar to hash tables.
Example below shows that memory is a list of bytes
and address is the key...

```
<'
 struct base_object {
  addr: byte;
  data: byte;
  };

 struct keyed_list {
   !memory : list(key:addr) of base_object;
   //writing to a keyed list
   write_memory(object:base_object) is{
     memory.add(object);
     print object;
    };

    //reading from keyed list
    read_memory(object:base_object) is{
     var local_object:base_object=memory.key(object.addr);
     print local_object;
    };
};
```




```
extend sys{
  data: base_object;
  olddata: base_object;
  mem_model : keyed_list;
  run() is also{
     //write data
     gen data;
     mem_model.write_memory(data);
     gen data;
     mem_model.write_memory(data);
     gen data;
     olddata=data;
     mem_model.write_memory(data);
     gen data;
     mem_model.write_memory(data);
     gen data;
     mem_model.write_memory(data);
     mem_model.read_memory(olddata);//retrieve old data
  };
};

'>
```



Another example

```
<‘extend sys{
  !cl: list(key:it) of uint(bits:4);
  run() is also{
     var ch: uint(bits:4);
     var i: int(bits:4);
     for i from 0 to 10{
          gen ch;
          cl.add(ch);
         };
     if cl.key_exists(8) then {
        print cl;
        print cl.key_index(8);
      }
    };
 };
‘>
```

Creating Struct Subtypes with when

- Creates a conditional subtype of the current struct type, if a particular field of the struct has a given value
- Struct members defined in the when construct can be accessed only in the subtype, not in the base struct

Syntax

- when type-qualifier'field-name base-struct_type
  {struct-member;….
  };
- base-struct-type: parent structure
- Field-name: The name of a field in the base struct. Only Boolean or enumerated fields are used
- type-qualifier: one of the legal values for the field named by field name

Explicit when construction

```
<'type Reg_n:[REG0, REG1, REG2, REG3];
type  Instr_type:[imm,reg];
type Dest_type:[mm_1,reg];
struct instr{
   %op1: Reg_n;
  kind:Instr_type;
  dest: Dest_type;
  when REG0'op1 instr{
      print_op1() is{
         out("instr op1 is REG0\n");
     };
   };
  when reg'kind instr{
      print_kind() is{
         out("Instr kind is reg\n");
     };
   };
};
'>
```

Implicit When Construction

```
<' type packet_kind:[transmit,receive];
    struct packet{
       kind:packet_kind;
       when transmit packet{
          length: int;
          print() is{
            outf("Packet length is %d\n",length);
           };
        };
    };
'>
```

Only visible when kind == transmit

Question: How do you access these subtype variables and methods?

Extending methods defined in base type

```
<‘ struct operation{
     opcode : [ADD,ADD3];
     op1: uint;
     op2: uint; result:uint;
     do_op(op1:uint,op2:uint): uint is{
          result=op1+op2;
      };
   };
```

Extending …

```
extend operation{
  when ADD3’opcode operation{
    op3: uint;
      do_op(op1:uint, op2:uint):uint is also{
        result=result+2;
       };
    };
  };
```

If the method is already described in the base struct
it should have the same number of arguments
in the when extension

Extending methods not defined in the base type

```
<‘ struct operation{
     opcode:[ADD,ADD3];
     op1:uint;
     op2;uint;
    };
   extend operation{
    when ADD operation{
       do_op(op1:uint,op2:uint):uint is{
         return op1+op2;
       };
     };
    when ADD3 operation{
       do_op(op1:uint,op2:uint):uint is{
       return op1+op2+op3;
       };
     };
   };
‘>
```

Can have different parameters and return types

Units

- Units were introduced in the e-language for easy portability
- Like structs they are compound data types that contain fields, methods and other members
- Unlike structs, a unit instance is bound to a particular component in the DUT (an HDL path)
- Also, each unit has a unique and constant e-path

Units vs Structs

1. The decision of whether to model a DUT component with a unit or a struct often depends on your verification strategy.

2. You intend to test the DUT component both standalone and integrated into a larger system. Modeling the DUT component with a unit instead of a struct allows you to use relative path names when referencing HDL objects.

3. Your e program has methods that access many signals at runtime. The correctness of all signal references within units are determined and checked during pre-run generation. If your e program does not contain user units, the absolute HDL references within structs are also checked during pre-run generation. However, if your e program does contain user units, the relative HDL references within structs are checked at run time. In this case, using units rather than structs can enhance runtime performance

So, why structs?

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

In short…

- Units are a kind of struct which can be….
  - Bound to any particular component in the DUT
  - Has a unique and constant parent unit (a static e-path) generated during pre-run generation.
  - In short units are static and structs are dynamic
  - Hence structs are used for data items, whereas units are to describe environment topologies and configurations

Syntax

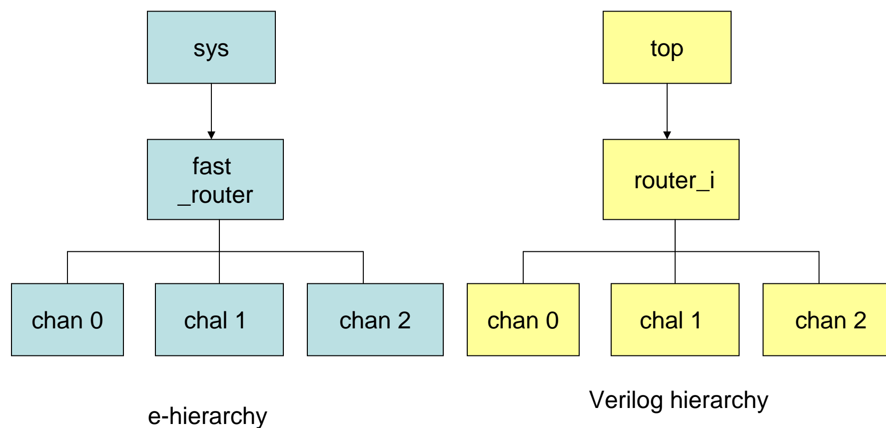
- unit unit-type {

  unit-member;

  …

  };
- Unlike structs, units can also have verilog members

Example



Binding the unit

```
<'
 unit fast_router{
   debug_mode:bool;
 };
extend sys{
  unit_core: fast_router is instance;
  keep unit_core.hdl_path()=="top.router_i";
  keep unit_core.debug_mode==TRUE;
};
'>
```

HDL path for the channels

```
<'
 unit router_channel{
  };
 unit fast_router{
   channels: list of router_channel is instance;
   keep channels.size()==3;
   keep for each in channels{
  .hdl_path()==append("chan",index);};
 };
'>
```

Constraining Generation

Constraints

- Test generation is a process producing data layouts according to a given specification. Constraints are statements that restrict values assigned to data items by test generation.
- A constraint can be viewed as a property of a data item or as a relation between several data items.

You may add constraint using…

- **Explicit constraints :** This are those declared using the keep statement or inside keeping {...} block.
- **Implicit constraints :** This are those imposed by type definitions and variable declarations. Implicit constraints are always hard.

Types of constraints

1. **Value constraints :** This restrict the range of possible values that the generator produces for data items, and they constrain the relationship between multiple items.
2. **Order constraints :** This influence the sequence in which data items are generated. Generation order is important because it affects the distribution of values and the success of generation.

- Both value and order constraints can be hard or soft:

1. **Hard constraints :** This (either value or order) must be met or an error is issued. This can not be over ridden.
2. **Soft value constraints :** This suggest default values but can be overridden by hard value constraints.
3. **Soft order constraints :** This suggest modifications to the default generation order, but they can be overridden by dependencies between data items or by hard order constraints.

You can define constraints in many ways:

- By defining a range of legal values in the field or variable declaration
- By defining a list size in the list declaration
- By using one of the keep construct variations within a struct definition
- By using a gen...keeping action within a method

Example-1

```
<'
struct constrain_gen_ex1 {
  // Explicit constrains
  x : int[1,3,5,10..100]; // is the same as
  x : int; keep x in [1,3,5,10..100];
  // Implicit Constrains
  l[20] : list of int; // is the same as l : list of int; keep l.size()==20;
  // Value constraints
  // Limits the address from 0 to 1024
  addr : uint[0..1024];
  // Read = 0 and Write = 1
  rd_wr: bool;
  // Weights for rd_wt command
  rd_wt: uint[0..100];
  wr_wt: uint[0..100];
•
```

```
// Order of generation is different
  // rd_wr was declared before rd_wt and wr_wt
  keep gen (rd_wt, wr_wt) before (rd_wr);
  // Generation based on weight
  // Soft Constraint
  …
  // Hard constraint
  data : uint;
  keep data != 0xdeadbeaf;

  // List constraint Example
  payload : list of byte;
  keep payload.size() < 10;
};
'>
```

Example-2

```
<' extend my_struct{
     keep x>4 and x!=6;// can use and, or
     keep x==y+25; //use addition, subtraction
     keep z==TRUE; //TRUE, FALSE
     keep y==method(); //can use methods
     keep x in [5..10,20..30]; //constrain variables to ranges
     keep x not in [1..3,5..8]; //Not in a range
     keep for each (p) in packets{ //packets is a list of packet
        p.length < 10; //each packet's length field < 10
      };
     keep packets.size()==50;//can use list pseudo-methods
};
'>
```

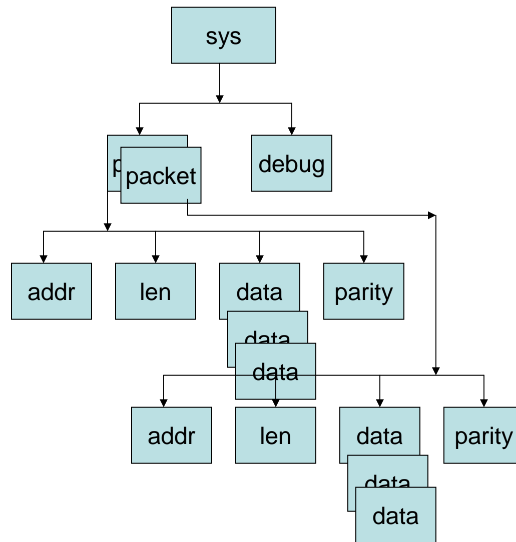
Implication Constraints

- Remember implications in 'e'
- X => Y means ***(not X or (X and Y))***
- Examples:

```
struct packet{
  size packet_sizetable;
     keep size == SHORT => len < 10;
     keep size == MEDIUM => len in [11..19];
     keep size == LONG => len < 20;
};
```

Order of Generation

```
<' struct packet{
    addr: uint(bits:2);
    len: uint(bits:2);
    data: list of byte;
    parity: byte;
  };
extend sys{
   packets: list of packet;
   debug: bool;
};
'>
```



Implicit Generation

- Constraints involving a method call:
  - keep parity==calc_parity(data);
- List slicing
  - keep z==my_list[x..y];
- Arithmetic operations
  - keep z=x*y;

Explicit Generation Order

```
<'struct packet{
  addr: uint;
  len : uint(bits:4);
  };
 extend packet{
  keep len==5 => addr < 50;
  keep gen (len) before (addr);
};
'>
```

What happens if we do not do this ? Discuss in the lab…

Constrain Resolution

- The generator must satisfy all the relevant constraints defined in:
  - The original struct definition
  - Extensions to the struct definition
  - Other struct definitions using the hierarchical path notation

Soft Constraints

- Specifies default preferences; to be over-ridden later
- keep soft
- If a soft constraint conflicts with a hard constraint it is over-ridden
- If it contradicts with other soft constraints, the last loaded constraint prevails

Example

```
<'
  struct cons{
 x: uint;
 keep x in [1..10]; //loaded 1st
 keep soft x > 3;//loaded 4th
 keep soft x == 8;//loaded 3rd
 keep soft x < 6;//loaded 2nd
};
'>
```

Resetting Soft Constraints

```
<' struct packet{
     len : uint;
     keep soft len in [64..1500];
    };
extend packet{
    keep len.reset_soft();//on a per field basis
    keep len > 2000;//apply a hard constraint
};
'>
```

Weighted Constraints

```
struct instr{
    opcode: cpu_opcode;
    keep soft opcode == select{
       30 : ADD;  //30/60 probability of selecting ADD
       20 : ADDI;
       10 : [SUB..NOP];
     };
};
'>
```

Procedural Flow Control

What is a Procedure?

- Interacts with the DUT
- Drives and samples the signals at the appropriate time
- Required to create interfaces, compute values, and act upon the fields of a struct or unit

Methods

- Syntax:
  method-name ([parameter-list]) [:return-type] is{
    action;…};
- Procedures can only be defined inside methods
- e-method is an operational procedure containing actions
- method can only be described inside a struct

Method Definition

```
<'
struct Packet{
   addr: uint(bits:2);
   zero_address() is bool{
     if(addr==0) then{
        out("Packet has address 0");
        result=TRUE; //result is an implicit variable
        } else{           //which holds the return value
        result=FALSE;
      };
 };
'>
```

Local Variables

- Inside a method, variables are declared with the var action
  - var count:int; //default is 0
  - var b_list: list of byte; //default is empty
  - var legal:bool;//default is FALSE

Values Accessed in method

```
<‘ struct C_struct{
      len: uint; //fields within local struct
      legal: bool; //---------do---------------
      d_struct_i : D_struct; //instantiation of a struct
      legal_length(min_len:uint):bool is{
          var count: int;
          if(len>=min_len) then{
            result=TRUE;
            count+=1;
           }else {
            result=FALSE;
           };
```

Arguments

Results (return value)

```
if (count==1) then{
    d_struct_i.addr=0;
    out("Incremented Counter");
    };
   };
}; //end of C_struct
struct D_struct{
  addr:uint;
};
'>
```

Fields inside other struct

Invoking Methods

- <' extend C_struct{

```
        post_generate() is also{
         legal=legal_length(64);
         };
    };
   '>
```

Methods are not executed until they Are invoked

Extending Methods

```
<‘ struct meth{
     m() is {
         out(“This is…”);
       };
     };
    extend meth{
       m() is also{
          out(“This is also”);
        };
     };
```

- ```
  extend meth{
   m() is first{
     out(“This is first”);
   };
  };
  extend meth{
     m() is only{
      out(“This is only…”);
     };
   };
  ‘>
  ```
  ```

Conditional Actions

- if-then-else
  - Syntax
    - if bool-exp [then] {action;..}
      [else if bool-exp [then] [action;…}
      [else {action;…}];

Example

```
• <‘ struct test1{
    a: int;
    b: int;
    meth1() is {
      if a > b then {
        print a,b;
       } else {
        print b,a;
       };
     };
  };
```

```
• struct test2{
    a_ok: bool;
    b_ok: bool;
    x: int;
    y: int;
    z: int;
    meth2() is{
      if a_ok{
         print x;
      } else if b_ok{
         print y;
      }
```

```
    else {
      print z;
      };
    };
};
'>
```

Case Action

```
<‘ struct packet{
      length: int;
    };
    struct temp{
      packet1: packet;
      meth() is {
         case packet1.length{
                64: {….};
                [65..256]:{…};
                default: {…};
          };
      };
 };
‘>
```

For loop

```
<‘ struct temp{
     a: int;
     meth() is {
       for i from 2 to 2*a do{//i is not needed to be
           out(i);                    //declared
        };
       for i from 1 to 4 step 2 do{
          out(i);
        };
```

```
  for i from 4 down to 2 step 2 do{
     out(i);
    };
  };
};
'>
```

For Each Loop

- ```
  <'
   extend sys{
    do_it() is{
     var numbers:={8;16;24};
     for each in numbers{
         print index;
         print it;
       };
  ```
  ```

```
var sum: int;
  for each(n) in numbers{
    print index;
    sum+=n;
    print sum;
   };
 };
};
'>
```

Other Constructs

- While
- Repeat
- Output Routines
  - out
  - Outf
  - print
  - do_print()

Events and Temporal Expressions

Temporal Expressions

- Timing & Synchronization are important when e and HDL processes communicate
- Temporal constructs are used to express and specify properties which vary with time
- Event Driven

Events

- Events define occurences of certain activities in Specman or HDL (verilog)
- Events can be attached to temporal-expressions (TEs)
- Can be unattached also
- An attached event is emitted when the TE succeeds
- Syntax: **event** event-type [is [only] TE];

Examples

```
<‘ struct m_str{
    event start_cnt;
    //unattached event, event manually emitted
    event top_clk is fall(‘~/top/r_clk’) @sim
    //sim is the sampling event (remember call back)
    event stop_cnt is {@start_cnt; [2]}@top_clk;
    //emitted when start_cnt is followed by 2 top_clk
    event clk is rise(‘~/top/cpu_clk’) @sim;
    event sim_ready is change(‘~/top/ready’) @sim;
};
‘>
```

Event Emission

- Events can be emitted explicitly or implicitly
- Use “emit” construct
- emit [struct-exp.]event-type;
- Does not consume time

Example with Emit

```
<‘ struct xmit_recv{
      event rec_ev;
      transmit() @sys.clk is{
         wait cycle;//wait for the next emission
         //of sys.clk event
         emit rec_ev;
         out(“Rec emitted”);
        };
```

```
receive() @sys.clk is{
  wait @rec_ev;//wait for the next rec_ev
                 //event, wait stops the TCM
                 //until the TE succeeds
   out("rec_ev occurred)";
   stop_run();
   };
```

```
run() is also{
 start transmit();//start two parallel processes
 start receive();//at 0 simulation time
 };
};
extend sys{
  event clk is @sys.any;// Finest granularity of time
   in Specman
  xmtrcv_i: xmit_recv;
 };
```

Redefinition

```
<'struct m_str{
   event m_str;
   event top_clk is fall('~/top/r_clk') @sim;
   event stop_ct is {
     @start_ct; [1]} @top_clk;
   };
  extend m_str{
   event stop_ct is only {@start_ct; [3]}@top_clk;
};
'>
```

Event stop_ct now redefined to specify an event which is implicitly emitted when the event start_ct is followed by 3 occurences of the event top_clk

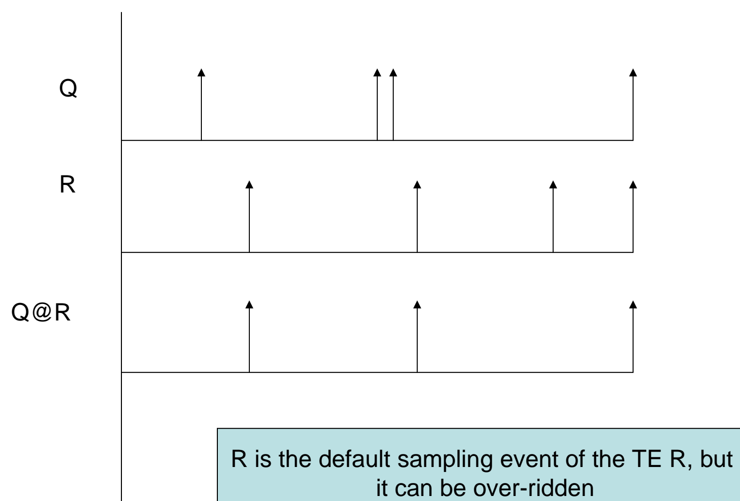
Sampling Events

- Events are used to define sampling points
- The TEs are evaluated at these sampling points
- The sampling period is the interval of time from the emission of a sampling event to the next time the sampling event is emitted
- All event emissions within the same simulation time is "simultaneous"

Certain Points

- Thus we see that Q@R means “evaluate Q every time the sampling event R is emitted”
- Q@R is a success (at the event R) if Q has been emitted since the previous emission of R
- Sampling Period for the TE includes the last occurence of the sampling event

An Example



Temporal Expressions

<‘ struct m_str{

event a_event is rise(‘~/top/start’) @sim;

event b_event is rise(‘~/top/end’) @sim;

event clk is rise(‘~/top/clk’) @sim;

event unary_e is @b_event @clk;

//b_event is a temporal expression. unary_e occurs at clk event when b_event occurs in a sampling period

- event boolean_e is true(‘~/top/clear’==1) @clk;

//Emit when TE is true at the rising edge

- event edgep_e is rise(‘~/top/a’==1) @clk;

//emit when the rise TE finds that a has gone from 0 to 1 in the sampling period

//is fall and is change are also possible

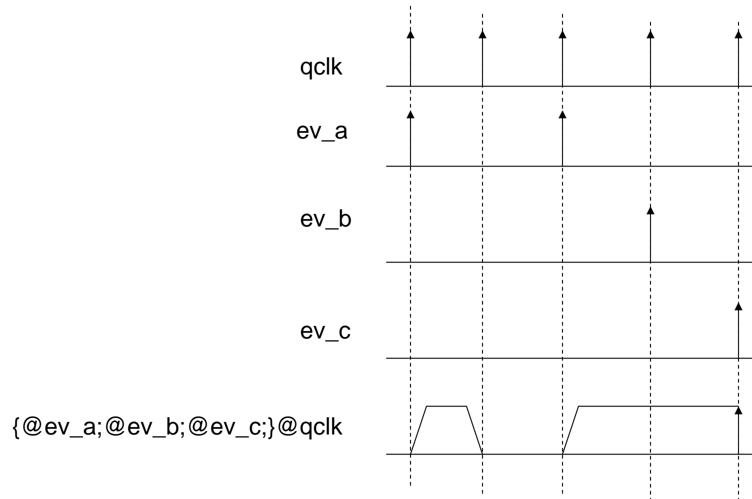
Delay and Cycle

```
temp_oper_example() @clk is{
    wait delay (10); //waits for 10 simulation
                          //time units
    wait cycle; //waits for the next emission of
                  //clk
   };
};
```

Sequence operators

- ; signifies a series of TEs over successive emissions of a sampling event
- Each TE following a ; starts evaluating in the sampling period, which comes after which the preceding TE succeeded
- The sequence succeeds when the final expression succeeds
- If any one misses, the sequence rolls back

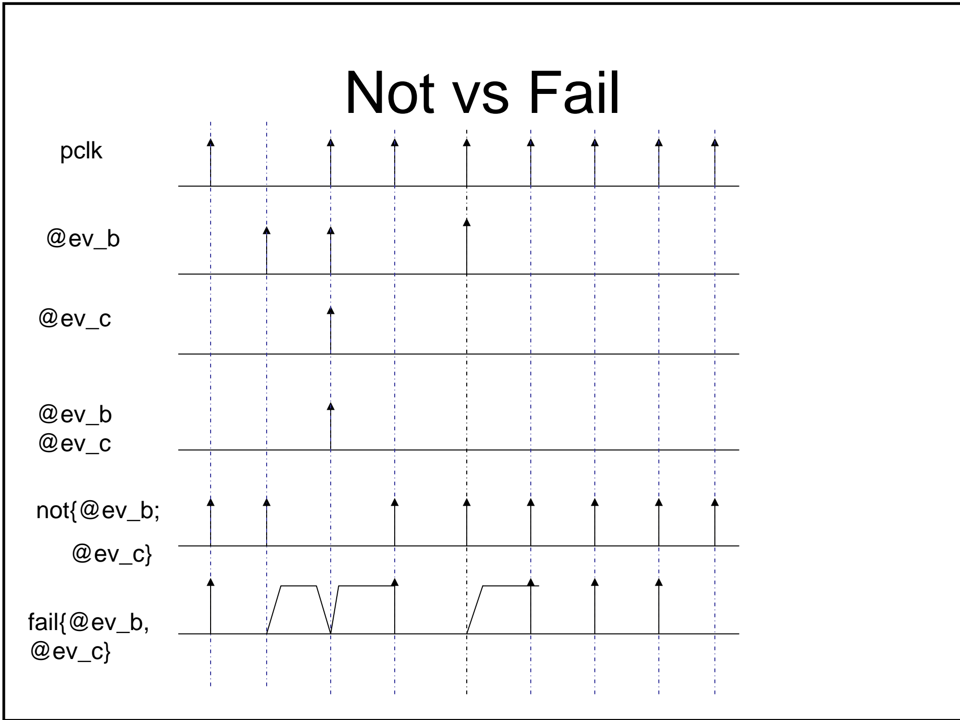
Example for Sequences



Not & Fail Operators

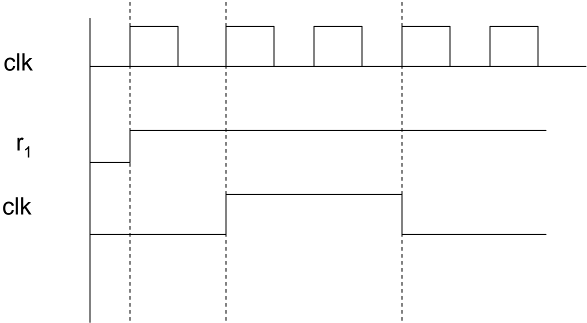
- event ev_d is {not{@ev_a;@ev_b;@ev_b}@ev_c}@clk
  - ev_d is emitted whenever there is an emission of ev_c which is not preceded by a TE, which is generation of ev_a and 2 succesive generations of ev_b

- event ev_d is fail{@ev_b;@ev_c};
  - ev_d is emitted if either of the following holds:
    - Event ev_b does not occur in the first cycle
    - ev_b succeeds in the first cycle, but ev_c is not emitted in the second cycle



Questions???

1. Can we express using the e-temporal constructs Property 1 of the Arbiter?
   1. LTL Statement: $G[\, r_1 \Rightarrow Xg_1 \wedge XXg_1]$



A Possible Code

```
event clk is rise('~/top/clk') @sim;
//Synchronize with DUV
event r1 is rise('~/top/req1') @clk;
//sampling event is clk
event g10 is {@r1; [1]} @clk;
event g11 is {@g10;[2]} @clk;
event g1 is {g1 and g2}@clk;
```

Checking for Property1

```
event gi0 is rise('~/top/g1')@clk;
event gi1 is fall('~/top/g1')@clk;
event property1;
expect property1 is
   @r1=>{@g1;[2];@g2}@clk;
```

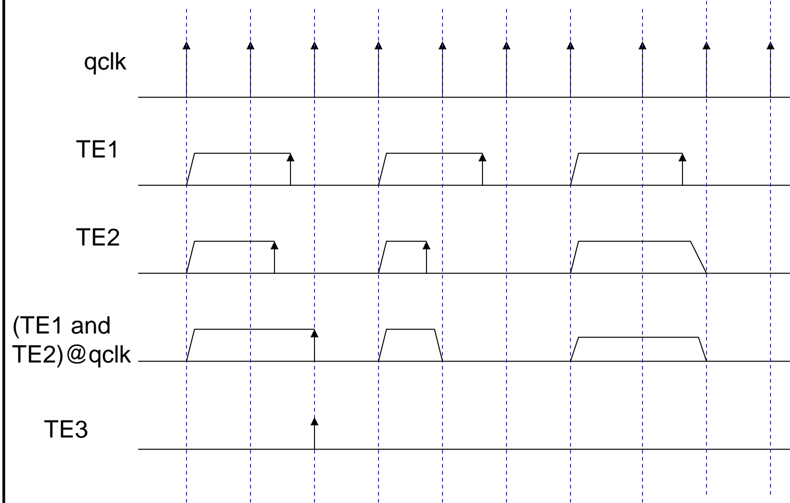
Tutorial-3 (contd.)

- Write e-snipets to express and check properties 2 and 3 of our arbiter.
- Compare the code size of your procedural construction with that using temporal constructs.
  Appreciate why 'e' is yet another language.
- Think of scenarios or designs (at least one) in which synchronization is important and try to express them using e-codes

And Operator

- The temporal **and** succeeds when both temporal expressions start evaluating in the same sampling period and succeed in the same sampling period.

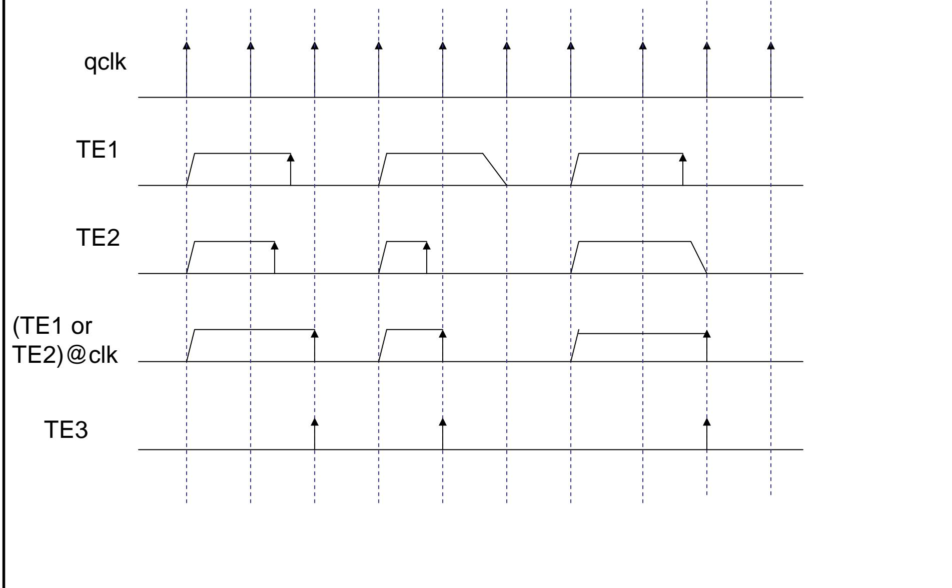
Example



Or operator

- The or temporal operator succeeds when either temporal expression succeeds. An or operator creates a parallel evaluation for each of its sub-expressions.

Example



Fixed Repeat

- wait {@ev_a ; [3]*@ev_b; @ev_c} @clk;

--wait action proceeds after ev_a, three successive occurrences of ev_b and ev_c

First Match Variable Repeat

- wait {@ev_a;
  [0..2]*@ev_b;@ev_c}@pclk;

// proceeds after any one of the three sequences:

1. {@ev_a;@ev_c}@pclk;
2. {@ev_a;@ev_b;@ev_c}@pclk;
3. {@ev_a;@ev_b;@ev_b;@ev_c}@pclk

They succeed on the first success of the TE

Contd.

- wait {[1..]*@ev_a; @ev_b}@pclk;

//proceeds after more than 1 events of ev_a, followed by 1 event ev_b at the next pclk event

- wait {[..3]*{@ev_a; @ev_b};@ev_c}@pclk;

//proceeds after between 0 and 3 occurrences of the sequence {@ev_a;@ev_b}, followed by the emission of ev_c at the next pclk event

True Match Variable Repeat

- They work on multiple occurrences of a TE from a lower to upper bound. They succeeds every time the TE holds.

event TE1 is {@reset; ~[3..5]} @pclk;

//succeeds three pclk cycles after reset, again at four pclk cycles after reset, again after five pclk cycles after reset

Eventually and Yield

- {@ev_c; @ev_a; eventually @ev_b}@pclk;

//TE succeeds when ev_c is followed by ev_a in the next cycle, and then ev_b sometime later

// Used to indicate that the TE should succeed some future time

- expect request => {[..99];@ack}@clk;

//The TE succeeds when ack is emitted after 1 to 100 cycles after request event, used for checking

Best of Luck

Quiz 1 on 20.2.07

(Any Updates will be posted on the web-page)