Specman Primer For SystemVerilog Users

Thorsten Dworzak

Verilab GmbH
thorsten.dworzak@verilab.com

April 2020

Part I

Specman Primer For SystemVerilog Users

A guide for knowledgeable SystemVerilog (SV) users who need to use Specman and the **e** language on their job but already have an understanding of constrained-random verification. Looking beyond just the syntax, you will find that there are more similarities than differences between the two languages. Some concepts are unique to the **e** language, and will be given specific attention. This guide is the culmination of many informal project training and coaching sessions given since 2013.

The **e** language is IEEE standard 1647 [IEEE1647]. There may be more widespread use of **e** in the future because it is the foundation of the DSL that Cadence implemented for their portable-stimulus (PSS) tool.

Note: This guide does not replace, but complements the Specman documentation, with the specific aim of helping SV users ramp-up faster and more efficiently.

Contents

I Specman Primer For SystemVerilog Users 1

1 The e Language 4
1.1 Structs and Units 5
1.1.1 Templated Types 7
1.2 Fields and Methods 7
1.3 Generation (Randomization) 8
1.4 Generation Constraints 9
1.5 Selected Data Types 11
1.5.1 Scalars 11
1.5.2 Collection Types 12
1.6 Aspects and AOP 13
1.7 Types of Inheritance 15
1.7.1 Like-inheritance 15
1.7.2 When-inheritance 16
1.8 Specman Phases 18
1.9 Ports - Connecting to the DUT 18
1.9.1 Signal-level Connections 19
1.9.2 Method Ports 20
1.9.3 TLM Ports 21
1.10 Events and Temporal Expressions 22
1.11 Concurrency Actions 24
1.12 Checks and Errors 25
1.13 Messaging 25
1.14 Coverage 27
1.14.1 Coverage of Events/TEs 28
1.15 Macros 29

II Specman Primer For SystemVerilog Users 30

2 Building Blocks of a Testbench 30
2.1 Static Components: Agents et al 31
2.2 Dynamic Components: Sequences and Items 32
2.2.1 Virtual Sequences 33
2.2.2 Sequence Example: Issuing Multiple Sequences in Parallel 34
2.3 Sequence Driver and BFM interaction 35

Copyright ©Verilab 2020

3 About Files **36**
3.1 What is a Testcase? . . . 36
3.2 eVC Directory Structure . . . 38
3.3 eDoc . . . 39

4 The Specman Tool **39**
4.1 Simulator Integration . . . 40
4.2 Interpreted vs. Compiled Mode . . . 41
4.3 Regression Run Management and Analysis . . . 41

5 Uncovered Topics **44**

6 About the Author **44**

1 The e Language

```
A hello world example.
<'
package helloworld;
extend sys {
    run() is also {
        out("Hello World!");
    };
};
'>
```

This example can be saved as file *test.e* and executed with

```
specview -p "load test.e; test"
```

This small example highlights already some distinctive traits of the **e** language:

- Any **e** code lives between the `<'` and `'>` delimiters. Everything outside of it is ignored.
- The `package` statement attributes all code in this module to a namespace (in **e**, by default everything is global).
- There is a top-level unit instance called `sys`. Put simply, units are the UVM components of the **e** language. `sys` is a pre-defined singleton. This is the root where any test environment is instantiated.
- Existing data types can be *extended*, i.e. data members and functions can be added to an existing type without definition of a new type.
- Every unit in **e** has certain phases which are implemented as callback functions. These functions can be overriden/extended (in this case, appended using `is also`). The most important phase-callbacks are:
  - `init()` - this is the constructor of an object.
  - `pre_generate()`, `post_generate()` - these are called before/after generation of an object (see below).
  - `run()` - this is started at time 0 of the simulation (if a simulator is attached), after all initial blocks in HDL have been executed.
  - `check()` - called after simulation has ended.

- The file gets *loaded*, i.e. Specman is an interpreter, while compilation is also possible. In fact, you can mix compiled and interpreted code. This allows to debug+fix something without requiring re-compilation.
- The *test* command is required. It calls all constructors and randomly generates all objects of the test environment.

There are more phases, we will get back to this topic later and also draw a comparison with SV.

Note that **e** does its own memory management, there is no destructor for an object. There is a `quit()` method of units/structs, but it is not a destructor. It is rarely used, but it is e.g. required if there are active temporal checks in a struct that prevent the struct from being garbage-collected.

The following chapters will explain these and other traits in more detail.

1.1 Structs and Units

These data types represent *Classes*. They are distinct mainly in that units have attributes `agent()` and `hdl_path()` which are needed to connect to a DUT. Units exist for the full lifetime of the simulation while structs are mostly used for dynamic data objects.

Consider the following example which defines a unit and a struct and instantiates them:

```
<'
-- define a new struct, inherit from "uvm_env_config"
struct env_config_s like uvm_env_config {
   !env: env_u
};

-- define a new unit, inherit from "uvm_env" and set
-- its hdl-path to the Verilog instance "testbench"
unit env_u like uvm_env {
   keep agent()     == "Verilog";
   keep hdl_path() == "testbench";

   -- instantiate the config-struct
   config: env_config_s;

   -- hook-up the config-struct
   connect_pointers() is also {
      config.env = me;
```

```
   };
};

extend sys {
   -- create an instance of the env unit
   env: env_u is instance;
};
'>
```

Here we are introducing a new phase, `connect_pointers()`, which is called just after `post_generate()` and which should be used to set-up pointers within the environment. It is comparable to the UVM-SV *connect_phase*. We use it here to pass a reference to its enclosing unit to the struct. This can also be obtained in the struct-context with the method `get_enclosing_unit(<unit-type>)`.

Both classes inherit from UVMe classes, but you may ignore this for now - we will come back to inheritance in **e** later.

Note the difference between the struct and the unit instantiation: the unit requires the `is instance` keywords. This is to indicate, that units have a constant place in the hierarchical tree. Thus you cannot modify the unit tree by generating unit instances on the fly, you can only add references to it at any place (like `env_config_s::env` in the example). Units can only be instantiated in units, which ensures that there is only one tree of unit instances starting at `sys`.

The struct in this example has the same lifetime as the unit, but structs can also be generated on-the-fly. Units are the fixed parts of the simulation environment and can be compared to *uvm_components*, while structs match *uvm_objects* in UVM-SV.

Using the `new` action, non-randomized struct variables can be created on the fly:

```
struct packet_s {
  header: header_s;
  data  : list of byte;
};

create_packet(payload: payload_s):packet_s is {
    var packet: packet_s = new with {
        .header = get_header();
        .data = pack(packing.low, payload);
    };
    result = packet;
};
```

Like in SV, the **e** language supports *interfaces* in the classical OOP sense (not to be

Copyright ©Verilab 2020

confused with the constructs used to connect to the DUT). An interface declares an API. The implementation is up to the struct that inherits the interface.

Remember: Use units to map to DUT module hierarchies, and use structs to model data/sequences/items/configuration.

1.1.1 Templated Types

If you want to generalize a class with respect to the type of its members, create a generic interface or a flexible numeric type, templates can be used. Example:

```
struct packet_s of (<data'type> : scalar) {
  header: header_s;
  data  : <data'type>;
};
```

When used, the template must be instantiated:

```
packet_inst: packet_s of (uint(bits:64));
```

The **e** language supports a number of constructs to make the templates type-aware. E.g. in the example we restrict the type parameter of the data field to be a scalar (numeric types and enums). We can also add conditions (boolean expressions) for bounding the values of a parameter.

1.2 Fields and Methods

Class data members are called *fields* and are random per default. We will get back to that in chapter 1.3. Fields can be declared `static` and/or `const`. Fields marked as *physical* using a % will get special attention in packing and unpacking.

```
struct foo {
    static !inst_id: int;     // define static member
    const %kind: kind_t;     // immutable and physical
    %data: uint(bits: 128); // declared physical
    !meta_data: uint;       // declared as non-generatable

    init() is also { inst_id += 1 }; // increment for each instance of '
       foo'
}
```

Member functions are called *methods*. In the **e** language the signature definition of a method and its implementation are in the same place (exceptions, see below). The following table shows the three different method declarations compared to SV:

Function type	Declaration	SV equivalent
time-consuming method (TCM)	`<fn-name>():<ret-type> @<event> is {... }`	task
regular method	`<fn-name>():<ret-type> is { ... }`	function
interface method	`<fn-name>():<ret-type>`	interface function

The TCMs are declared with a default sampling event - this is different to SV in that at the start the TCM is synchronized to this sampling event. It is also a convenience feature that allows to omit the sampling event in expressions. We will later see examples for this.

Instead of declaring the method with a `{...}` body, we can use `empty` or `undefined`:

- If the method body is `empty`, the function can be called but does not do anything and returns the default value for the return type. This is similar to a function declaration in C/C++.
- If the method body is `undefined`, it makes this a kind of an abstract class method (pure virtual method in SV).

1.3 Generation (Randomization)

In SystemVerilog, `randomize()` is a method of an object and can be called over and over again on the same object. In Specman, however, you cannot randomize an object without also creating it. The combined act of constructing and then randomizing a struct or unit is known as *generation*.

Randomization is at the heart of Specman and *the standard way to construct a struct or unit instance is to generate it.*

Because of this ubiquitous randomization, the normal way to configure a unit (a verification component) is to set constraints on the randomization of its fields. This has two important and interesting effects. First, you can get some genuine randomization of your testbench setup if that's appropriate. Second, and more commonly useful, configurations that are controlled by generation constraints are sure to have their values properly set up as soon as the unit is constructed, because it is created and randomized in a single hit. That's not quite true though! Specman does the creation

(the construction of a struct's memory footprint) first. It then calls struct or unit's `init()` method to complete its initialization; by extending this method you can customize the field initializations in the same way that you would in a SV object's `new` method. Finally, randomization takes place - and, as in SV, there is a `pre_randomize()` and `post_randomize()` hook that of course can be extended too.

In a SystemVerilog class, any data member that you want randomized by the class's `randomize()` method must be declared as random using the *rand* prefix in its declaration; ordinary data members are not randomized. Specman works the other way around - everything is randomized by default, and you must explicitly mark things as *non-generatable* (using an exclamation-point prefix on their declaration) if you don't want them randomized. This has an important side-effect for fields (data members) of struct type. Any field of a struct type *will* exist (i.e. will have been constructed) if it is not marked non-generatable. But non-generatable fields of struct type are *not* constructed automatically; Non-generatable fields of scalar type obviously don't need to be constructed, so they always exist, but they will be initialized to their standard initial value when their enclosing struct is constructed.

You can also generate things (fields of structs and units, unless they are units or ports) at any time during a simulation run, using the `gen` action (generation on-the-fly).

1.4 Generation Constraints

As in SystemVerilog, generation can be controlled using constraints that specify Boolean conditions that will be true after successful generation. The basic syntax is

```
keep <name> is [only] <expression>;
```

Naming of constraints is optional, but encouraged. It is especially useful for grouping constraints or overriding hard constraints, see further below.

Simple constraints work in a broadly similar way to SystemVerilog. There are a few superficial differences:

- the constraint implication operator is written `=>` in **e**, but it is `->` in SystemVerilog
- the **e** set membership operator is `in` (SystemVerilog uses `inside`) and you can also use `not in` with the obvious effect
- no curly braces around constraints in **e**

- distribution constraints in SystemVerilog are written using `dist`, but in **e** they use the `keep soft...select` syntax (note that SystemVerilog 2012 has clarified that dist constraints are also soft in SystemVerilog, just as they are in **e**)

And then there are a few major differences:

- A generatable field of struct or unit type is invariably constructed and randomized by e's generation operation, whereas in SystemVerilog if a data member of object type has the value `null` it will be ignored and will still be `null` after generation.

- Generatable list fields are always resized by randomization; this means you need to specify a constraint on the size of any list that will be randomized. Whereas in SystemVerilog a queue or dynamic array will be left with its pre-randomization size unless there is an explicit constraint on the array's `.size()` attribute.

- In **e**, constraints can be written that constrain the values of struct and (more commonly) unit references; this cannot be done in SystemVerilog. So, for example, it is common in **e** to see a constraint such as

  ```
  keep child_component.parent_ref == me;
  ```

  After `child_component` is generated by the parent unit, its `parent_ref` pointer will be a reference to `me`, its parent.

Constraints can be grouped using `all of`. This is particularly useful when a single condition has multiple implications, e.g. in constraining fields that are only present in subtypes of an instantiated unit. Together with naming the constraint, good style would be e.g.

```
keep c_agent agent is a ACTIVE vr_ahb_slave (aa)  => all of
{
  aa.bus_no      == master_bus_no + 1;
  aa.has_checks  == TRUE; // the == TRUE can be omitted
  aa.tolerant    == read_only(parent_p.tolerant); // a unidirectional
      constraint
};
```

Additional features not available in SV are *range-generated fields* and *table-constraints*, both of which are not explained here for brevity. The following table summarizes all of the above:

Construct	e Syntax	SV equivalent
implication (if LHS true, RHS applies)	=>	->
set/range membership	in	inside
soft constraints	soft	soft
if/else	ternary operator ?:	if { } else { }
generation order	keep before	solve before
distribution constraints	soft ... select	dist
list fields	lists are resized by randomization, i.e. need to constrain size too.	lists keep their pre-randomization size
iterative constraintes	for each	foreach
struct or unit references	can be constrained	
grouping constraints	all of {};	{} (mandatory syntax)
range-generated-fields	`rgf=`	
table-constraints	`in_table`	
type-constraints	`type ... is a ...`	

1.5 Selected Data Types

1.5.1 Scalars

Scalar data types: in **e** they are similar compared to the two-value SystemVerilog data types. Predefined types are inlineint/uint, bit, nibble, byte, time and enumerated types. Built-in type casting is either implicit (e.g. `uint` to `int`) or can be done explicitly using `<orgtype>.as_a(<newtype>)`. When declaring a scalar variable, you can add a range modifier, e.g.

```
var x: uint(bits: 34);
var y: int(bytes: 5);
var z: uint[0..4](bits: 4); // a number between 0..4, but using 4 bits
    as storage.
```

Enumerated types: they are ubiquitously used throughout Specman environments, because they are central to the *when-inheritance* of units/structs explained in chapter 1.7 below. Enumerated types can also be sized and assigned to numbers which makes it easier to map them to a DUT signal, e.g.

```
var color: [red, blue, green](bits: 2); // enum using 2 bits as storage
var car: [mercedes = 2, bmw = 4]; // enums with fixed integer
    representation
```

They can also be declared as type:

```
type car_t: [hyundai, kia];
...
var my_car: car_t;
```

Of course, the enumerated type is also extendable, i.e. values can be added to the original type. It is possible to access the values that can be assigned (introspection feature):

```
extend car_t: [audi, volvo];
...
print car_t.all_values(); // prints "hyundai kia audi volvo"
```

1.5.2 Collection Types

The **e** has not many collection types, mainly *lists* and *keyed lists* (hashes). Lists can be declared sized or unsized (but even the sized ones can be extended - comparable to dynamic arrays/queues in SV). They can also be multi-dimensional. There is a data type called `set` to handle uniquified collections of integers. Also, constant ranges are used ubiquitously.

If lists are generatable and declared unsized, a constraint for their size should be added (if not, a global, user-controllable Specman constant kicks in that prevents generation of huge lists).

```
struct packet_container_s {
  packets: list of packet_s;
  keep soft packets.size() == 10; // list with 10 elements will be
      generated with random structs of packet_s
  !primes[5]: uint; // list with fixed size
  init() is also {
    primes = {1; 2; 3; 5; 7}; // list is not generatable, so we
        initialize it.
  };
};
```

To guide the generation of lists, use a `for each` constraint. Let's extend the struct defined above:

```
extend packet_container_s {
  // calc header using a method "get_header()" and one of the values
  // in list "primes"
  keep for each in packets {
    soft it.countable == TRUE;
    // note the implicit variables "it" and "index"
    it.header == get_header(primes[index % primes.size()]);
  };
};
```

Lists have built-in methods, similar to functional languages. Some of these are higher-order functions and allow some functional programming. A list behaves like a SV dynamic array or queue, depending on the methods you use (`.pop()`/`.push()` vs. `resize()`). Here is an example that uses the `packets` list from above:

```
 // count number of packets that have field "countable" set to TRUE
var sum: uint = packets.all(it.countable).size();

 // the same but more efficient (no intermediate list)
var sum2: uint = packets.count(it.countable);

// make a new list composed of the result of the packet_s::get_crc()
// method for each element
var crcs: list of uint = packets.apply(it.get_crc());
```

Some of the list-methods can be used in bi-directional constraints, i.e. they don't enforce an input-only direction like other method-calls.

Here is a set example calculating the difference set of two sets:

```
var primes: set = [1,2,3,5,7,11,13];
var fibonacci : set = [1,2,3,5,8,13];
out(fibonacci.diff(primes)); // prints [7,8,11]
```

Basically, sets are just binary values with each 1-bit representing a set member.

1.6 Aspects and AOP

The **e** language is aspect-oriented (and also object-oriented, see next chapter). To quote the Wikipedia definition of Aspects [Aspects]:

> In computer science, an *aspect* of a program is a feature linked to many other parts of the program, but which is not related to the program's pri-

> mary function. An aspect crosscuts the program's core concerns, therefore violating its separation of concerns that tries to encapsulate unrelated functions. For example, logging code can crosscut many modules, yet the aspect of logging should be separate from the functional concerns of the module it cross-cuts. Isolating such aspects as logging [...].

Basically, any cross-cutting concern is an aspect. In **e**, the most important aspects are

- Stimuli generation
- Monitoring
- Checking
- Functional Coverage

But then, it can also be something specific like "hook-up all shadow registers to the physical bus monitors" or "connect a scoreboard to Bus A and Bus B".

To separate the aspects, **e** allows to modify struct and unit definitions and methods after their definition. That way you can keep specific aspects in their own files, the load-order determining how the resulting struct/unit or method is implemented. For structs/units and enum types, this is done by means of the `extend` statement.

Methods can be extended with `is only`, `is first`, or `is also` to override, prepend or append the original definition (and here the load-order of files is important!). Note that this is not the same as stacking a number of include-directives. Each extension has its own namespace for local variables.

```
<'
-- Define the unit.
unit agent_u like uvm_agent {
    flavour: flavour_t;
    display() is {
        print flavour;
    };
};

-- Add a new property to the existing struct definition.
extend agent_u {
    name: name_t;
    display() is also {
        print name;
    };
};
'>
```

Care should be taken when extending base types as this is not a reuse friendly practice. The AOP nature of **e** allows this anywhere in the load-list of the testbench environment which may lead to problems if the type is subsequently used. It is better to create a child class or extend a subtype - see next chapter. An exception of this rule are patches.

1.7 Types of Inheritance

The **e** language supports two types of inheritance: *when-inheritance* and *like-inheritance*.

1.7.1 Like-inheritance

Like-inheritance is what you know from SystemVerilog or other classic OOP languages. In SV you would derive a new class from an existing one like:

```
class my_class extends base_class;
```

Inheritance allows to define a new type that inherits all fields and methods from a base type. Example:

```
<'
unit env_u like uvm_env {
   display() is {
      out("in env_u ...");
   };
};
unit eth_env_u like env_u {
   display() is also {
      out(me, "... actually in eth_env_u");
   };
};
unit stack_env_u like eth_env_u {
   display() is only {
      out(me, " in stack_env_u");
   };
};
extend sys {
   eth_env   : eth_env_u is instance;
   stack_env : stack_env_u is instance;

   post_generate() is also {
      eth_env.display();
      stack_env.display();
```

```
    };
};
'>
```

Creates the following output:

```
Loading example5.e ...
Doing setup ...
Generating the test with IntelliGen using seed 1...
in env_u ...
eth_env_u-@1... actually in eth_env_u
stack_env_u-@2 in stack_env_u
Starting the test ...
Running the test ...
No actual running requested.
Checking the test ...
Checking is complete - 0 DUT errors, 0 DUT warnings.
```

What is noteworthy here is

- You can reference the instance of a unit/struct from inside with `me`. Printing `me` gives a unique id that you could also obtain from anywhere else using the global method `sn_unique_id_for_struct(<instance>)`. Printing actually calls the built-in `to_string()` method of a unit/struct.
- The `eth_env_u` only extended the `display()` method of its parent unit - that's why it still prints "in env_u". Whereas `stack_env_u` overloaded the whole method.
- Note that in **e** you never call the super class's methods - the method called is either the parent's method (if not extended in the child), a mixture of both (if extended in child with `is first` or `is also`) or just the child's method (if extended using `is only` in child). In that sense, all base-class methods are virtual.
- A method can be declared with the prefix `final` which prevents further extension/override.

The Specman data browser allows to display all extensions of a method.

1.7.2 When-inheritance

When-inheritance uses one or more member fields of a unit/struct (a boolean or enum type) - the *when-determinants* - to specify the subtype of a struct/unit. Example:

Copyright ©Verilab 2020

```
<'
type ahb_flavour_t: [A5, LITE];
unit ahb_env_u like uvm_env {
   -- We will use this field as when-determinant.
   const kind: ahb_flavour_t;
};

extend A5 ahb_env_u {
   -- This extends the A5 subtype
};

extend ahb_env_u {
   when A5 ahb_env_u {
      -- This also extends the A5 subtype, using a different notation
   };
   when LITE ahb_env_u{
      -- This extends the LITE subtype
   };
};
'>
```

Let's look at this more closely again:

- `uvm_env` is a pre-defined unit in **e**. It is the UVMe base-type for every unit, and can be omitted in the definition. See the Cadence documentation on what API it offers.
- The base unit `ahb_env_u` has a field `kind` of type enum that we can use as when-determinant. It is declared as constant because the unit hierarchy is constructed at generation time and then never changed anyway (this has memory advantages but is not required, you can change subtypes on-the-fly). Other than that there is no need to highlight this as a when-determinant.
- The first `extend` statement creates a subtype of the name `A5 ahb_env_u`.
- The second `extend` displays another syntax of implementing the subtypes; this can even be shortened by omitting the base-type: `when LITE { ... }`.

Note that when-inherited struct types are generated at run-time (dynamic typing) - in SV you must apply the factory pattern to mimic this functionality. There is no need for the factory singleton in **e**.

1.8 Specman Phases

The simulation lifetime is partitioned into a number of phases that apply to all units. The top-level unit `sys` is special in that some construction has to take place before anything else can happen, but you can refer to the Cadence documentation for more details. There are four main phases which are further divided into one or more callbacks that will be executed for each unit in the unit tree.The following table depicts the phases and is an interpretation of how this compares with UVM-SV.

Major Phase	**Valid for**	**Callback**	**UVM SV equivalent**
Generate (Setup)	unit, struct	init()	build_phase()
	unit, struct	pre+post_generate()	pre+post_randomize()
Generate (Elaborate)	unit	connect_pointers()	connect_phase()
	unit	connect_ports()	
	unit	check_generation()	
	unit	elaborate()	end_of_elaboration_phase()
Run	unit, struct	run()	start_of_simulation_phase(), run_phase()
Check	unit, struct	extract()	extract_phase()
	unit, struct	check()	check_phase()
Finalize	unit, struct	finalize()	report_phase()

The phase callbacks of every unit/struct can be overloaded with `is also`/`is first` (`is only` is not recommended!) to add custom functionality.

1.9 Ports - Connecting to the DUT

The Specman and the RTL simulation do not share the same process, which means connecting the testbench and the DUT (or just any component in a foreign simulation domain) is not as obvious as in Verilog/VHDL/SV. This is achieved by *ports*. There are different types of ports:

- Simple ports (for signal-level connections)
- Event ports (for event-based connections, i.e. Verilog events or signal changes)

- Method ports (for procedure calls)
- Buffered ports (not used anymore)
- TLM interface ports
- TLM 2.0 sockets

We will not cover all here. Let's look at an example for simple and event ports first.

1.9.1 Signal-level Connections

```
<'
extend encoder_bfm_u {
   keep agent()     == "Verilog";
   keep hdl_path() == "~/testbench";

   data: out simple_port of int(bits:64) is instance;
     keep bind(data, external);
     keep data.hdl_path() == "dut.data_i";

   clock: in event_port is instance;
     keep bind(clock,external);
     keep clock.hdl_path() == "clk";
     keep clock.edge() == MVL_0_to_1;

   drive()@clock$ is {
      wait;
      data$ = gen_data();
   };

   run() is also {
      start drive();
   };
};
'>
```

Here we define two ports in a unit. Let's have a closer look.

- We constrain the unit's `agent()` and `hdl_path()` attributes, which determines that the unit connects to a Verilog module with the absolute hierarchical reference `"~/testbench"`.
- The ports are declared as instances (like unit instances) which establishes their place in the unit/instance hierarchy.

- We need to define the port's binding, which is external here. The default binding is empty and in that case reads would yield 0 while writes would have no effect. Since the unit's agent_type is "Verilog", the port should connect to a Verilog signal. This example is using the declarative form, ports can also be bound using procedural do_bind() calls which should be done in the connect_ports() callback (which makes the bind statement above redundant).
- We assign an hdl-path to the ports. This implicitly sets the default binding to external.
- out simple_port ports usually drive Verilog reg types. It is possible to drive wires as well.
- The event_port binds to a signal clk and is sensitive to a low-high transition here. There is more on events in chapter 1.10.

Tip: reuse dictates that the port declarations and the assignments of hdl-paths are in separate files. The port instantation would be in an eVC (**e** Verification Component). Per convention, ports are defined in a unit called *port-map*, which acts as a container and allows to use relative hdl-paths for port connections. Every entity that needs to drive or monitor a port gets a pointer to this unit. Conceptually this resembles an SV interface/virtual interface. The actual hdl-path assignment would live in a user-file that *configures* the eVC for a particular HDL counterpart.

Note: if you need to trigger on the transition of a signal, e.g. for clock and reset, it is recommended to use event ports for this, rather than using a temporal expression on a simple port. Briefly, the reason for this is that the sampling of simple ports is problematic with the occurance of more than one event between two sampling events.

1.9.2 Method Ports

Method ports are an interface to a remotely defined method, e.g. in a foreign simulation domain. It is also possible to define method ports internally but it is rarely used. A typical usage is the connection of an eVC with a UVM SystemVerilog component (a more modern technique is to use TLM ports of course, as below). The following example creates a method port on the **e**-side to call the SystemVerilog function method frob_ml_monitor::send_item() to transfer an item of type frob_item_s.

```
Method port example
<'
method_type frob_mon_item_mt(item: frob_item_s);

extend frob_monitor_u {
```

```
    //------------------------------------------------------------
    // Hierarchical path of the corresponding UVM component in the UVM
       hierarchy
    keep external_ovm_path() == "monitor";

    // Class name of the corresponding UVM component in the UVM hierarchy
    keep external_ovm_class() == "frob_ml_pkg::frob_ml_monitor";

    // Method port to transfer collected items to the SV-side
    send_item: out method_port of frob_mon_item_mt is instance;
    keep bind(send_item, external);
    keep send_item.external_ovm() == TRUE;

    // Name of method on SV-side
    keep send_item.external_ovm_path() == "send_item";

    // Overload the monitor hook method
    handle_ti(di_arg: frob_item_s) is also {
       message(LOW, me, "MON write_data_sampled_hu: ", di_arg);
       send_item$(di_arg);
    };
};
'>
```

Note: This is not the complete code! Mapping the compound data type `frob_item_s` between the languages is done with support of the `mltypemap` utility that generates the necessary files (provided with the Specman installation).

Of course it is also possible to do the opposite, i.e. calling an **e**-method from an SV port. From the Cadence online documentation:

> Invocation of input method ports from a SystemVerilog UVM class is done using proxy classes. These proxy classes are created inside a SystemVerilog package called `specman_package`. This package is created as part of the Specman stubs file.

1.9.3 TLM Ports

Specman supports TLM and TLM 2.0 ports - this document does not cover TLM 2.0.

The TLM ports are a standardized means to connect to a foreign simulation. Let's have a look at an example for a port definition in both languages:

Define the port:

Specman/e	UVM SystemVerilog
put_port : out interface_port of tlm_nonblocking_put of simple_trans is instance	uvm_nonblocking_put_port (simple_trans) put_port

Define the export:

Specman/e	UVM SystemVerilog
put_export: export interface_port of tlm_nonblocking_put of simple_trans is instance	uvm_nonblocking_put_export #(simple_trans, consumer) put_export

Bind the port:

Specman/e	UVM SystemVerilog
put_export.connect(put_port)	put_export.connect(put_port)

Note: you can also use the aliases `interface_port/interface_imp/interface_export` instead of `<direction> interface_port`. We would not recommend this as the intent is less obvious.

1.10 Events and Temporal Expressions

Events are struct members and can be the result of a temporal expression or procedurally generated using `emit`. They can be compared to Verilog events, rather than to the UVM event objects. Each event implicitly defines a call-back method (see example). Temporal expressions define temporal behaviour and can be compared to SV properties.

Example:

```
Event example
<'
extend monitor_u {

   event trans_done_e;
   // define clock-event using reference to port-map
   event clk_e is @portmap.clk_rise_port$;

   // at start of sim, spawn monitor task
   run() is also {
      start monitor_task()
   };

   // monitor a signal in an endless loop
```

```
    monitor_task()@clk_e is {
       while TRUE {
          sync rise(portmap.ack_port$)@clk_e;
          emit trans_done_e;
          wait; // default sample event: clk_e cycles
       };
    };

    // implicit event call-back
    on trans_done_e {
       sample_trans();
    };

    // Temporal check with action block
    expect MON_SHUTDOWN is @alarm_e =>{ [..3]; @reset_e } @clk_e
       exec { me.quit() }
       else dut_error("reset must follow alarm within 1..4 cycles");

   // extend the event call-back method from above
   on_trans_done_e is also {
      display_trans();
   };
};
'>
```

This example demonstrates a few concepts:

- the event trans_done_e is emitted procedurally in the monitor_task() time-consuming method (TCM).
- the event clk_e is an alias for an event port that is defined in an instance called portmap. Event ports can be bound to foreign simulator objects and have an event edge() attribute that defines the trigger cause.
- the monitor_task() TCM, which has clk_e defined as the default sampling event, first waits for a rising signal bound to a port ack_port sampled at the clk_e event (which we would not need to specify here as it is the default).

  The sync as opposed to wait guarantees that execution is continued if an event happens in the current tick. The wait consumes at least one cycle of the sampling event.
- The subsequent wait statement defaults to one cycle of the sampling event of the TCM, e.g. wait [4] means 4 cycles.
- The on <block> struct member executes a block right after its event triggers.

It may not contain any time-consuming actions itself. This pseudo-method can also be extended with e.g. `on_<event-name> is also { ... }` (note the _).

- The `expect|assume` struct members are like assertions in SV. They can contain action blocks for passing and fire error messages when failed.

  This one states that within 1..4 cycles after an alarm event, a reset event should follow upon which the monitor unit quits itself (which also ends the monitor task thread).

  The `=>` implication operator is identical to the SV one. To model the SV `->` operator, which states that the RHS expression occurs in the same cycle of the sampling event, you need to work with the Specman `detach` operator. Refer to [IEEE1647] for more temporal operators.

Of course, event definitions can be overriden with `is only` in later modules, much like methods. Here though, `is first` or `is also` would not make sense.

There are a number of built-in events, like `session.start_of_test`, `sys.any`, `sys.new_time`, but we leave it to the reader to explore any further using the Cadence documentation. Built-in events are handy to synchronize with the foreign simulation. Because there is not a single simulation kernel, it can be tricky to run the Specman and the RTL simulator processes side-by-side.

It should be mentioned that there is a callback mechanism to trigger on ports. It is rarely used and we recommend a more procedural programming style that keeps the number of threads low.

1.11 Concurrency Actions

TCMs and action-blocks can be executed (*spawned*) in parallel to the current thread.

Purpose	**e**	**SV**
Spawn a single task	`start <tcm>()`	`fork <task()/block>` `join_none`
Spawn parallel threads, continue when all have ended	`all of { ... }`	`fork begin ... end join`

Spawn parallel threads, continue when first has ended and kill all others	`first of { ... }`	`fork begin fork`
		...
		`join_any disable fork;`
		`end join`

Copyright ©Verilab 2020

There is also a feature to spawn a number of tasks determined during run-time. Example:

```
all of for each (seq) in (sequence_list) {
    do seq on p_driver keeping { .id == index };
};
```

This will start all sequences at the same time-step and continue execution (because `start_sequence` is non-time-consuming).

1.12 Checks and Errors

The **e** language clearly separates between DUT errors and user/programming errors. The `check` command is used for checking the DUT behaviour. Unexpected behaviour can be flagged using the `dut_error[f]()` method. E.g.

```
check LIMIT_CHECK that (a < config.limit)
  else dut_error("bug found!");
```

The use of named checks, `LIMIT_CHECK` in this case, allows for mapping checks to verification plans in the Cadence tools.

User and programming errors are reported via `warning()`/`error()`/`fatal()`. There is `try` to catch programming errors, though it is not a fully-fledged exception mechanism known from other programming languages. Example:

```
try {
  var rfield := rf_manager.get_struct_of_instance(me).get_field("
      type_plate");
  var ret_rft := rfield.get_type().get_name();
} else {
  error("Error during initialization of reflection interface")
};
```

Use `assert()` for checks. *Note:* this cannot be disabled like the Verilog `assert` statement, so it is safe to use.

1.13 Messaging

This chapter only describes the unit-tree based messaging mechanism to control message verbosity. The logger-based mechanism is deprecated.

Notification messages are issued using the message command, e.g.

```
<'
extend message_tag: [MONITOR];
...
message(MONITOR, LOW, "message text"){
  // Optional action block to print more information;
  // has same verbosity as message
  out("...");
};
'>
```

which specifies the verbosity of the message, an optional tag `MONITOR` of the enum type `message_tag`, and an optional message block. Per default the `NORMAL` tag is used. Tags and verbosity are used for filtering of messages. The Specman command `set_message [options] <unit-exp>` controls which messages of a particular unit or unit-tree (the command is recursive by default) are shown. There is also a procedural interface available via the `message_manager` singleton. Example:

```

Messaging example
<'
-- Create a new message tag
extend message_tag: [CHANNEL_MESSAGES];

-- Define a unit
type c_t: [FOO, BAR];
unit channel_u like any_unit {
  -- when-determinant
  kind: c_t;

  run() is also {
     message(CHANNEL_MESSAGES, MEDIUM, append(" This is instance ", kind
         , "."));
     message(CHANNEL_MESSAGES, LOW, "run called for instance.");
  };
};

extend sys {
  -- Instantiate the units
  c1: FOO channel_u is instance;
  c2: BAR channel_u is instance;

  elaborate() is also {
    -- In the elaborate phase, we are sure that c1 and c2 exist
    message_manager.set_screen_messages(sys.c1, CHANNEL_MESSAGES, LOW);
    message_manager.set_screen_messages(sys.c2, CHANNEL_MESSAGES, FULL);
```

```
  };
};
'>
```

So we have set different verbosity levels for the two units. This creates the following output:

```
Starting the test ...
Running the test ...
[0] FOO channel_u-@1: run() called for instance.
[0] BAR channel_u-@2: run() called for instance.
[0] BAR channel_u-@2:  This is instance BAR.
No actual running requested.
Checking the test ...
Checking is complete - 0 DUT errors, 0 DUT warnings.
```

1.14 Coverage

Defining Coverage groups: Coverage group definition in **e** is done by using the `cover` construct which is equivalent to the `covergroup` model in SV. The data for sampling is an `item` which corresponds to the `coverpoint` element in SV. The name of a coverage group must be the name of an event type defined previously in the struct - similar to when using a trigger event for the SV `covergroup`. Here is an example of a basic coverage definition:

```
<'
extend dut_config_u {
  event dut_config_e;
  burst_length: uint[1..16];

  get_clk_frequency(): uint is { ... };
  is_clock_enabled(): bool is { ... };

  cover dut_config_e using text="DUT configuration", global is {
    item burst_length;  // covers enclosing unit's field
    item freq: uint = get_clk_frequency() using ranges = {
      range([1..100], "Low");
      range([100..MAX_UINT], "High");
    },
    illegal = (freq < 1), when = (is_clock_enabled());
  };
};
'>
```

Note that in this example the item `burst_length` is a member of the enclosing unit whereas `freq` is defined in the cover group. But this is like in SV.

Differences and Similarities to SV: If a cover group is defined in **e** it is automatically created, no need to call `new` explicitly. Cover groups in the **e** are struct/unit members and are even less like real objects than in SV (i.e. you cannot pass around handles or instantiate the same cover group twice).

Procedural sampling using `covers.sample_cg(...)` is also supported. This uses the `covers` singleton - there is no sample method of the cover group itself!

Like in SV coverage groups and items can specify a number of options to control the coverage collection and refine the generated buckets for a coverage item. Cover groups can be extended/overriden which allows easy refinement of coverage models.

1.14.1 Coverage of Events/TEs

If TEs are used in expect/assume statements, they are covered if the Specman configuration option `collect_checks_expects=TRUE` is set. The statement should be named though. Example:

```
expect CHECK_INT_SEEN is true(TRUE)@interrupt_rise_e;
```

`CHECK_INT_SEEN` will appear in the collected coverage. The other possibility is to create a cover group of the event:

```
cover interrupt_rise_e is {
  item irq: bool = TRUE using ignore = (not irq);
};
```

Alternatively, the configuration option `auto_cover_events=TRUE` allows to globally cover all events (in a covergroup `session.events`), but this is time/memory-consuming and not recommended.

In SV, coverage for checks can be achieved by emitting events for successful checks or - better - using named immediate assertions for checks, like in

`my_check_name: assert (my_condition).`

1.15 Macros

The **e** language itself is extensible. There are three types of macros:

1. *define*: Simple text replacement, e.g. `#define ADDR_WIDTH 10`
2. *define-as*: syntactic macro for text replacement with arguments; while it is possible to pass macro parameters to Verilog macros, the difference is that in define-as macros the arguments and the macro itself are checked by the parser. The macro is assigned a syntactic category, like *action* and can only be used in the context where the category is allowed.
3. *define-as-computed*: syntactic macro with arguments constructing replacement string using e-code. That is, the macro executes e-code that builds a string which is returned by the macro to get the actual text replacement. This is a quite powerful concept.

Example for a define-as macro:

```
<'
-- Macro for a Ruby-like for-each loop
-- Example:
--    list.each {print it};

-- Define the macro with the syntactic category 'action';
-- the string in double-quotes is the match-expression
define <vlab_list_each'action> "<exp>\.each <block>" as {
  for each in <exp> <block>;
};
'>
```

Usage example for this macro:

```
<'
extend sys {
  run() is also {
      var l: list of uint = {1;2;3;4};
      var f: uint = 1;
      l.each {
         f = f * it;
      };
      print f; // prints 24
  };
};
'>
```

Example for a define-as-computed macro:

```
<'
-- Macro that creates a string from unquoted characters (like the
-- qs() in Perl)
-- Example:
--    keep hprot.hdl_path() == qs ( master0_hprot );

define <vlab_util_string'exp> "qs\(<any>\)" as computed {
   var rs: string= str_expand_dots(<any>);
   result = quote(str_trim(rs));
};
'>
```

By specifying the syntactic category of the macro, true language extensions are possible. Be aware though, that the macro is part of the global namespace. If you need to avoid that, look at Templates (see chapter 1.1.1).

It should be mentioned that there is also the *Table* mechanism that captures regular code structures (statements, struct members, constraints, etc.) which are parameterized. It allows to separate the code from its set of parameters which can be in a .csv or .xls file.

Part II

Specman Primer For SystemVerilog Users

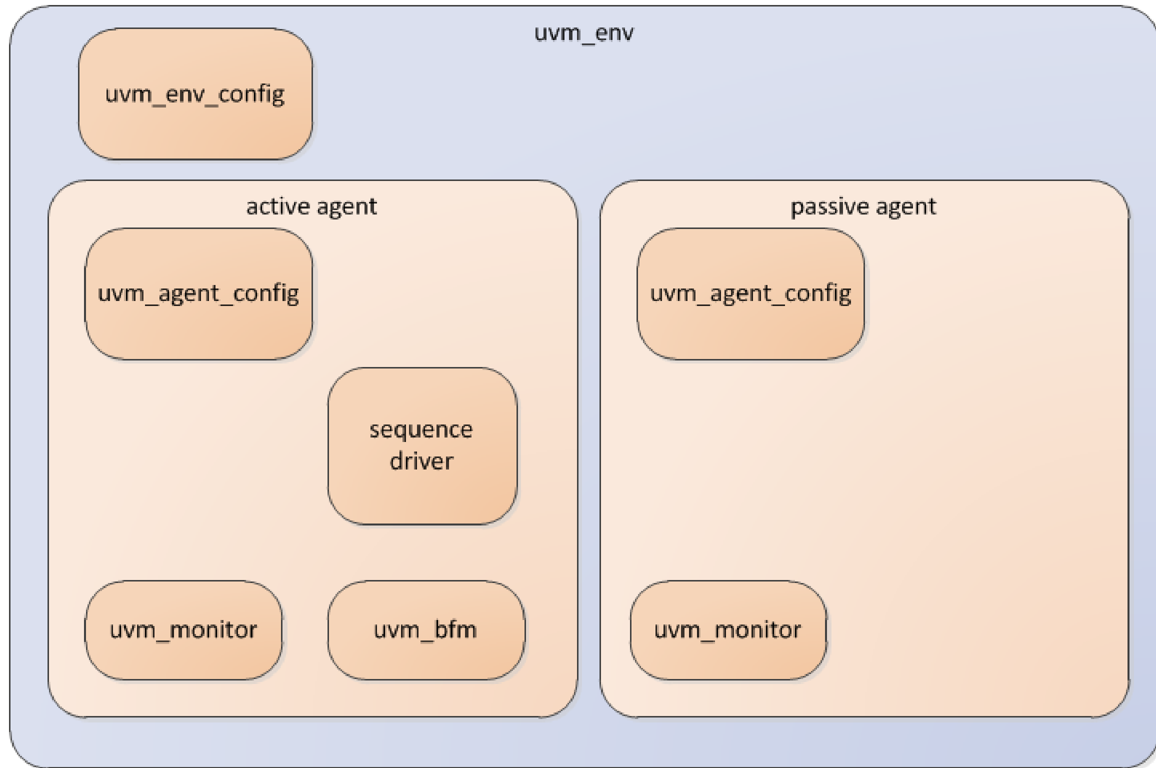
This second part focuses on how to apply the language to test and testbench construction. It will also cover some tool aspects.

2 Building Blocks of a Testbench

The building blocks of Specman test environments are very similar to SystemVerilog environments, because the OVM (the UVM predecessor from Cadence) was substantially based on the eRM (**e** Reuse Methodology). In SystemVerilog the BFM is called driver, whereas the sequence driver is called sequencer - but that's where the differences end. These days Specman has a new base class library called UVMe which

aims to achieve some consistency with the UVM-SV. The UVMe is only a "thin" layer above the eRM. The eRM is still accessible.

The following diagram depicts the units that a UVMe environment is composed of (you have seen a similar diagram a million times before, but we will not spare you).



In this example the environment has two agents, one active and one passive. Conventional reuse dictates that the BFM and the monitor are completely decoupled. Note that the sequence driver does not have a base class - of course it has one, but for reasons explained below you don't need to care. There are some more base classes, like `uvm_collector` (a low-level monitor), `uvm_signal_map` and the struct `uvm_config_params` .

2.1 Static Components: Agents et al

The components in UVMe are `uvm_env`, `uvm_agent`, `uvm_collector`, `uvm_bfm`, `uvm_monitor` . They are just listed here for completeness. As they have similar counterparts in the UVM-SV we will not further elaborate on this.

Copyright ©Verilab 2020

Note that the eRM base-classes are prefixed with any_. The generic base classes are still being used, like any_unit, any_sequence etc.

2.2 Dynamic Components: Sequences and Items

The dynamic components of the **e** instance hierarchy comprise sequences and items which are usually generated on-the-fly. Sequences may encapsulate other sequences or data items. Sequences are defined via a macro, which in its simplest form it looks like this:

```
sequence my_seq using item = my_item_s;
```

This sequence statement creates the following:

- A new enumerated type called my_seq_kind.
- A new sequence struct, which inherits from the predefined any_sequence.
- A new sequence driver unit my_seq_driver, which inherits by default from any_sequence_driver .

The macro has more optional parameters which allow you to pick different names for the sequence and its setting. Note that the item struct my_item_s must already be defined (and inherits from any_sequence_item). It gets automatically extended with a reference to the sequence driver.

Note: If the item option is admitted, a *virtual* sequence is declared.

A sequence is first like any other struct, but it has two fields, driver and kind, which dictate that it is used only inside a sequence. Isn't that a chicken+egg problem? No, because together with the sequence macro mentioned above, you get a set of when-inherited types of the base sequence, namely MAIN, SIMPLE, RANDOM. MAIN is the root of the sequence tree, therefore used to generate all other sequences of the same type. Here is an example using the sequence we created above.

```
<'
-- Create a new sequence kind
extend my_seq_kind: [SUB];

extend SUB my_seq {
   id: uint;
};

-- Extend the MAIN seq to do what we want
extend MAIN my_seq {
```

```
   -- The sequence instance we are about to generate
   !sub: SUB my_seq;

   -- Overload the MAIN sequences body() method
   body()@driver.clock is only {
      for i from 1 to 4 {
         do sub keeping { it.id == i };
      };
   };
};
'>
```

The `do` statement generates the sequence object `sub` (using the supplied constraints) and hands it over to the sequence driver. Note that it was not necessary to constrain the driver field, because the default is the same as the parent sequence. It is not obvious when the do statement finishes, because it depends on the sequence driver/BFM interaction. Potentially this is a problem where `sub` gets overwritten by a do performed before the sequence driver has consumed the previous one.

Note: an advanced feature of Specman are the `item_constraint` and `sequence_export` macros which allow to constrain fields piercing through several hierarchies of items/sequences. This is useful if a high-level sequence wants to directly constrain fields in a low-level item or sequence.

For the reverse application, i.e. reaching up from a constraint in an item or sequence, **e** provides the `in_sequence()` and `in_unit()` constructs (*tree constraints*).

Refer to the Cadence documentation for a detailed view of the sequence/sequence-driver/BFM API.

2.2.1 Virtual Sequences

Virtual sequences are sequences with no inherent item. That's it. Full-stop.

No, really, what it implies is that virtual sequences *do* other virtual sequences or sub-sequences (or a mix thereof). At the end of this hierarchical relationship is usually[1] an item that is dealt with by a BFM (in SV: driver). To do this, a virtual sequence knows about all sequence drivers of the sub-sequences it needs to create.

[1]True, but in a layered sequence architecture, you have BFMs that pass sequence items to the sequence drivers of lower layer sequences.

2.2.2 Sequence Example: Issuing Multiple Sequences in Parallel

To start a number <n> of sequences in parallel on <n> sequence drivers, it is possible to *do* these sequences in parallel threads using the `all of` statement of the **e** language.

```
body()@driver.clock is {
  all of {
    { do seq1 keeping { .driver == driver1 } };
    { do seq2 keeping { .driver == driver2 } };
  };
};
```

However, this only works if <n> is known at compile time. If you have a list of sequences, you can use the `all of for each` statement.The following example shows how to start <n> threads and wait for their completion. This uses the `any_sequence::start()` method.

```
<'
// Starting sequences in parallel and sync'ing on their ended-event
extend vr_ad_sequence {
  !finished: bool;
  on ended { finished = TRUE };
};

extend my_virt_seq_s {
  !reg_seqs: list of vr_ad_sequence;
  n: uint;
  keep n in [1..4];

  body()@driver.clock is {
    for i from 1 to n do {
        var seq: vr_ad_sequence;
        gen seq keeping {
           .driver == ...
           ...
        };
        reg_seqs.add(seq);
    };
    all of for each (reg_seqs) {
        do it;
    };

    sync true(reg_seqs.count(it.finished) == reg_seqs.size());
  };
};
'>
```

Sequences as member variables should be declared non-generatable. It can have unintended consequences if they are created during the initial generation phase.

It is important to always constrain the `driver` and `kind` field of a sequence (the kind is declared as `const`, so you cannot change it during the lifetime of the object).

2.3 Sequence Driver and BFM interaction

When a sequence does not generate a sub-sequence, but an item, the BFM appears on stage. You must define a method of the BFM that takes an item as argument and drives it to what-ever-is-at-the-other-end.

```
<'
-- Add a driving method to our BFM
extend my_bfm_u {
   -- define a clock event, using a reference to a portmap.
   event clk is rise(pmp.clock$)@sim;

   drive_it(drive_item: my_item_s) is {
      pmp.valid$ = ~pmp.valid$;
      pmp.head$ = pack(packing.low, drive_item);
      -- wait one cycle to be ready for the next item
      wait;
   };
};
'>
```

Next, you have to decide whether the interaction between sequence driver and BFM is PUSH or PULL. The latter is the default, meaning the BFM pulls items from the sequence driver and drives them. We will only give an example for PULL mode here:

```
<'
-- PULL mode example
extend my_bfm_u {
   -- This thread pulls items from the driver.
   execute_items()@clock is {
      var seq_item: my_item_s;
      while TRUE {
         -- Query the driver - this stalls the BFM until
         -- there is an item. You can also use try_next_item()
         -- if you want to keep control to the BFM.
         seq_item = driver.get_next_item();
         drive_it(seq_item);
         emit driver.item_done;
      };
```

```
   };

   -- Pin wiggling
   drive_it(item: my_item_s)@clock is { ... };

   -- Call our endless thread at the start of the simulation.
   run() is also {
      start execute_items();
   };
};
'>
```

The decision whether to use PULL or PUSH mode depends a bit on what you want to do if sequence driver and BFM are idle, i.e. no user sequences are generated. The idle time can be filled by the sequence driver with default sequences (PUSH mode) or by the idle BFM with default items (PULL mode).

3 About Files

Files are also called modules. The load order of modules is important, whereas the order of declarations within a module is not.

3.1 What is a Testcase?

The testcase is the module loaded last. It imports every other module that is part of the testbench/test scenario. Typically, it consists of an import statement, and the extension of a sequence that exhibits the test procedure. Example:

```
<'
-- Import the testbench configuration
import include/scc_srmd_config;
-- Import additional features
import scc_limitations;

-- Extend the main sequence to only execute one SRMD sequence
extend MAIN scc_main_s {
   !test : SRMD scc_main_s;    // declare as non-generatable
   body() @driver.clock is only {
      do test;
   };
};
```

```
-- Constrain the virtual sequence used in this test
extend SRMD scc_main_s {
   keep soft drs == select {
      30: WR;
      70: RD;
   };
   keep nr_of_bursts in [2..50];
};

-- Additional constraints valid for this test only
extend scc_env_u {
   keep has_scoreboard == TRUE;
   keep end_of_test_cycles == 2000;
};
'>
```

This is a typical testcase. It is quite short, because all the underlying infrastructure that is needed to execute it is implemented in our test environment. Typically, in UVMe the test environment is instantiated in a file denoted *configuration*. There may be one or more configurations.

This is different to the UVM-SV tests where the test usually instantiates the test environment (at least, the base-class of the tests does). So the UVM-SV test base-class can also be interpreted as UVMe configuration, depending on the architectural choice made for the UVM-SV testbench. Also, in UVM-SV typically all tests are compiled up-front, whereas in UVMe only the test to be run is loaded.

Let's look at the different sections of this example:

- Import statements: import the verification environment used in this test (and probably by a range of other tests). Other imports are test-specific.
- Extend the `MAIN` sequence. It is usually the root sequence executed by the sequence driver. Here, we overload the `body()` method of the sequence to do our own sequence. The `body()` method of a sequence is a time-consuming method where the sequence does what it does. It triggers on the `clock` event of its sequence driver (of which every sequence has a reference called `driver`).
- Extend the `SRMD` sequence that we execute in this test. This is test-specific tweaking to verify a certain feature.
- Finally, the test might require to configure some parts of the environment. This is done by extending existing structs/units. As the test is the top-most file loaded, it is okay here to extend a type without creating a new when/like inherited type.

Tip: it is recommended to keep all temporary workarounds and constraints in a separate file imported by the tests (here, called scc_limitations.e). All code should be documented with Bugzilla/JIRA issue keys or what is applicable in your case. The AOP paradigm enables this technique nicely. This way it is easy to (manually) check whether this file is empty before tape-out.

3.2 eVC Directory Structure

The eRM and the UVMe define a standard directory structure for verification components. The top-level directory is named after the eVC, here <evc> as placeholder. The directory should contain at least:

```
<evc>/
    e/
    examples/
    docs/
    PACKAGE_README.txt
```

Note that the PACKAGE_README.txt is standardized and machine-readable. E.g. contains the version of the eVC and the versions of sub eVCs that this one is dependent on. See the Cadence documentation for more details.

The file-naming is also (per recommendation) standardized:

Guideline	**Details**
Prefix for all eVC source files	<evc>_
The eVC top file always located at (imports all other files)	<evc>/e/<evc>_top.e
Standard names for aspect files	An appropriate name for the aspect, e.g.: types, checker, cover, monitor, sequence
Natural domain names for entities	Examples are master, slave, burst
Uniform order/prefix/suffix of names (agent first, feature last)	<evc>_master.e, <evc>_checker.e <evc>_master_checker.e, <evc>_slave_driver.e, <evc>_master_monitor.e, <evc>_slave_bfm.e
User-owned files	<evc>_<config>_config.e, <test-name>.e

The <evc>_<config>_config.e file instantiates the top-level environment in sys and configures the environment for the given purpose (active/passive, hook-up to a DUT, etc.).

Tip: it is recommended to have a file named `<evc>_env_h.e` or `<evc>_agent_h.e` which defines all the components that comprise an environment/an agent, but leaves the detailed implementation for other files. That way the testbench topology can be understood at a glance.

If you want to have a closer look at this aspect, we recommend David Robinson's book about AOP [Robinson]. It devises a strictly aspect-centric naming scheme for files. A file that hooks up a register-model to an AXI bus (thereby linking two aspects) would be called e.g. `<evc>_axi-register-connect.e`.

3.3 eDoc

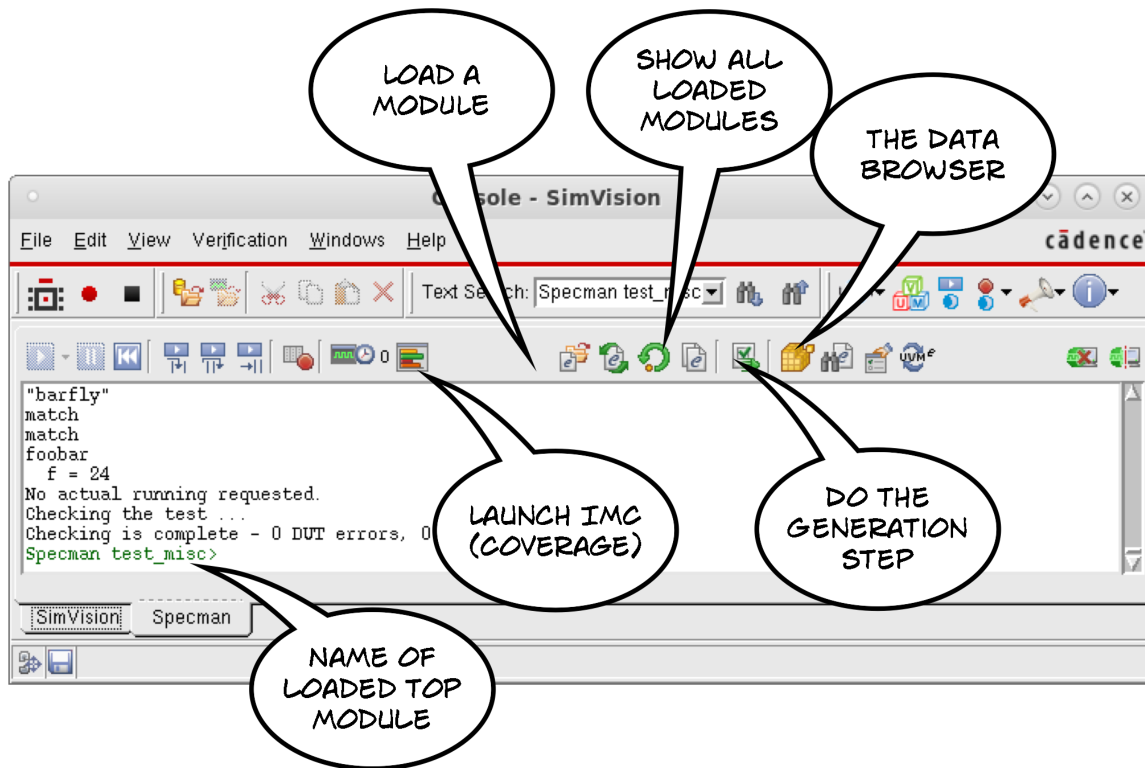
Specman provides an automatic documentation generator like Doxygen. You can find it in the Specman GUI in Tools− >Methodology Utility. There are also some other visualization utilities, e.g. for register-models.

4 The Specman Tool

From the simulator's perspective, Specman is a tool in the passenger seat, connected via PLI. These days the integration with the simulation host is tighter, but the interaction mode is the same. You can fire up specview without specifying a simulator, you can also run it on its own, executing **e** code if you like[2].

The following figure shows the GUI in stand-alone mode:

[2]If you run Specman without its scheduler, it does not even require a license and can be used as general purpose programming language



The main features of the Specview GUI are highlighted. The prompt shows the top-module, usually the test. IMC is the unified coverage GUI for all Cadence tools. The data browser shows the Specman test environment hierarchy allows to inspect everything that is a part of it (structs, units, member variables, events, methods etc.).

4.1 Simulator Integration

If we had launched the simulator, the GUI displayed here would also show a tab for the simulator console, next to SimVision and Specman.

The typical simulation flow is:

- Call the simulator (xrun) with the compiled snapshot (DUT plus HDL test-bench) and load the top-level **e** file that imports the rest of the environment.
    - The simulator is at time 0, Verilog initial blocks have been executed.
- Do the generation step (click GUI button or call test from the command-line): this generates the instance hierarchy of your testbench with seed 1 (default).

- You can inspect the instances using the Data Browser.

- Switch to the simulator console and start the simulation.

4.2 Interpreted vs. Compiled Mode

Any **e** code can be run interpreted or compiled or both. This is especially helpful in debugging, because you can execute any code from the specview commandline. Here are a few examples:

```
Specman test> print {2;3;4;5}.reverse()
  {2;3;4;5}.reverse() =
0.    0x5
1.    0x4
2.    0x3
3.    0x2
Specman test>

Specman test> break on line 98 @ahb_bfm if (sys.time > 14000 ns)

Specman test> print scoreboard-@345.queued_items.all(it.
match_type == UNMATCHED).sort(it.end_time);
```

The commandline also allows to call any methods of the loaded environment, displayed in the last example. Files can be compiled using options for the `xrun` command. Typically, the portion of the code that is imported first and does not change (e.g. imported eVCs, libraries) is compiled. Everything else loaded on top of that is executed in interpreted mode. Interpreted vs. compiled is as usual a trade-off between run-time and turn-around time (not so much a matter of debuggability because that has improved a lot in recent years). If the files you are working on are interpreted, its easy to do debug - reload - test - debug and so on.

Batch files: files that contain specman commands and/or **e** code and have the suffix `.ecom` can be executed from the prompt using `@<file>` (without the suffix).

4.3 Regression Run Management and Analysis

Two additional tools exist. *VManager* is both ...

- a regression runner. It uses plain-text `.vsif` files as input and creates `.vsof` files with pass/fail data.

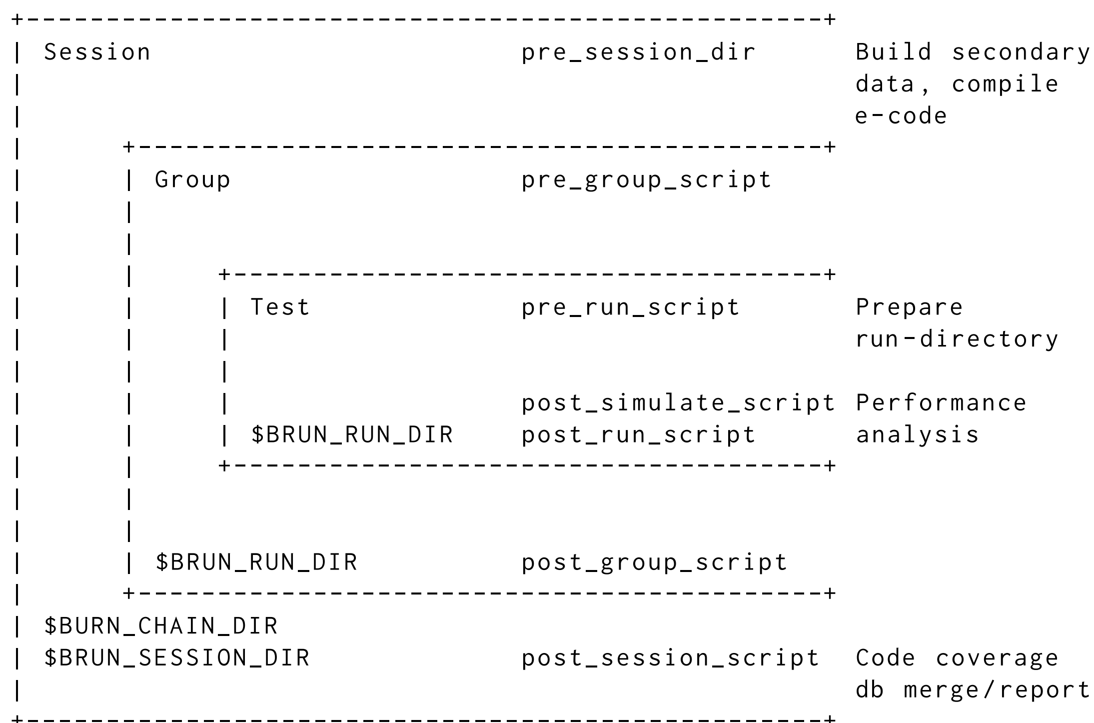
- a test analysis tool. Tests can be graded, sorted by error, run-time, etc. Functional and other coverage from the regression (`.ucdb` files) run can be mapped to a verification plan. The IMC coverage analysis tool is also integrated.

VManager uses a client/server model with a PostgreSQL database.

The runner has a hierarchical view of the regression. At the top is the session, containing groups, containing tests. Each container has certain attributes (like a timeout value for the runs or the name of a test) that are inherited from the enclosing container or are overwritten by the container itself. The `.vsif` files are controlling these attributes.

The following diagram depicts this relationship together with the scripts that the user can specify for each stage, the typical use-case for a script, and the most important Unix environment variables available for a container:

vManager Container Scripts



The names `pre_session_script` etc. are attributes of the container.The user can define new attributes that are available to the tools as Unix environment variables `$BRUN_<attribute-name>`.

Note that a regression can also comprise of non-Specman runs, like directed tests or formal analysis runs. This is achieved via setting the appropriate attributes in the `.vsif` files. Example:

```
session vm_test {
    top_dir : $ENV(MY_REGRESSION_AREA);  // where to put the regression
       data
    pre_session_script: scripts/pre_session.sh; // creates the snapshot

    // user defined attributes! see vm_attributes.csv
    e_compile   : yes;                    // no|yes;
    nc_coverage : B:E:T;                  // B:E:T:F:U|All|none
    output_mode : terminal;               // open an xterm in pre-
       session
    // DRM settings
    #include "scripts/drm_settings.vsif.inc"
};
// SPECMAN based tests
group random {
  pre_run_script: <text>scripts/pre_run.sh</text>;    // sets up the
     simulation
  run_script:     <text>scripts/run_script.sh</text>; // runs the
     simulation

  // NOTE: post_run_script is run __AFTER__ the scan script.
  // You cannot catch errors in there.
  // If you need to parse for errors in post simulation
  // commands, use post_simulate_script
  post_run_script: <text>scripts/post_run.sh</text>;  // cleans
     simulation
  // Filter script that searches STDOUT of the simulation
  scan_script: "vm_scan.pl $ENV(VMANAGER_HOME)/bin/ies.flt \
                $ENV(VMANAGER_HOME)/bin/shell.flt" ;
  count: 1;// 50;      // how many runs
  timeout: 180;        // 0 means NO timeout
  seed: random;        // which seed

  test test1 {
    pre_commands : <text>sn_do_something</text>; // to
       SPECMAN_PRE_COMMANDS
    top_files: smoke.e;                           // e test file to load
  };
};

// directed HDL tests
group directed {
    count: 1;
```

```
    test directed1 {
        run_script: <text>scripts/directed_run1.sh</text>;
    };
    test directed2 {
        run_script: <text>scripts/directed_run2.sh</text>;
    };
};
```

VPlanner is started from within VManager. It is used to create a verification plan by manually mapping coverage groups and items to features (*forward annotation*). It replaces the legacy XML verification plan that had features which were *back-annotated* to coverage groups/items by placing tags in the **e**-code. Coverage can also be code-coverage, pass/fail status, checks, or coverage from a formal tool, as long as a `.ucdb` database is created by the respective tool.

5 Uncovered Topics

We could not cover everything in this tour-de-force. For further reading, some missed topics are listed here:

- Reflection interface. Specman supports a rich API to support meta-programming, which is the ability of a program to inspect and change itself.
- Packing and Unpacking, i.e. conversion/serialization of scalars, collection types and structs
- Objection mechanism (typically used for end-of-test handling but allows user-defined objections just like in UVM)
- Testflow. This supports run-time phases (a mechanism that also exists in UVM-SV but is hardly used in either methodology).
- Foreign language interfaces (TLM-2, DPI-C, Python, etc.)

Have fun!

6 About the Author

Thorsten Dworzak received his Dipl.Ing. in Electronic Engineering from the University of Siegen (Germany) in 1997. He has been working in digital design and

verification of ASICs for the consumer and embedded market for 20+ years. He is currently a Principal Engineer with Verilab's verification consulting team.

References

[IEEE1647] "IEEE Standard for the Functional Verification Language e", IEEE Std 1647-2019, Aug 2019

[Robinson] David Robinson, "Aspect-Oriented Programming with the e Verification Language", 1st Edition, Morgan Kaufmann, 17th August 2007

[Aspects] https://en.wikipedia.org/wiki/Aspect_(computer_programming), Wikipedia, April 2020

Copyright ©Verilab 2020