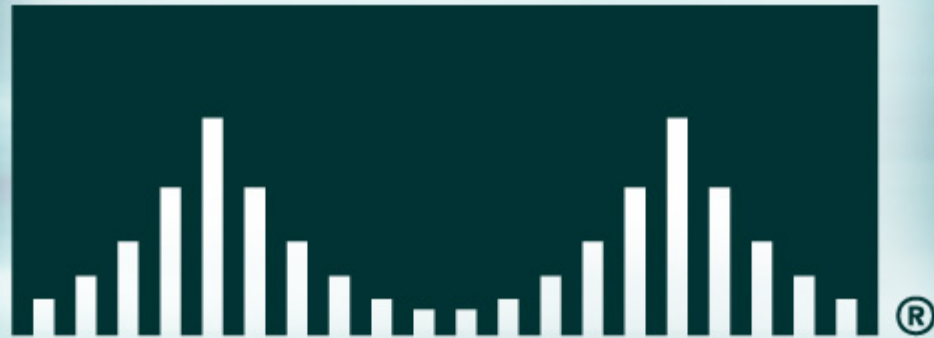


CISCO SYSTEMS

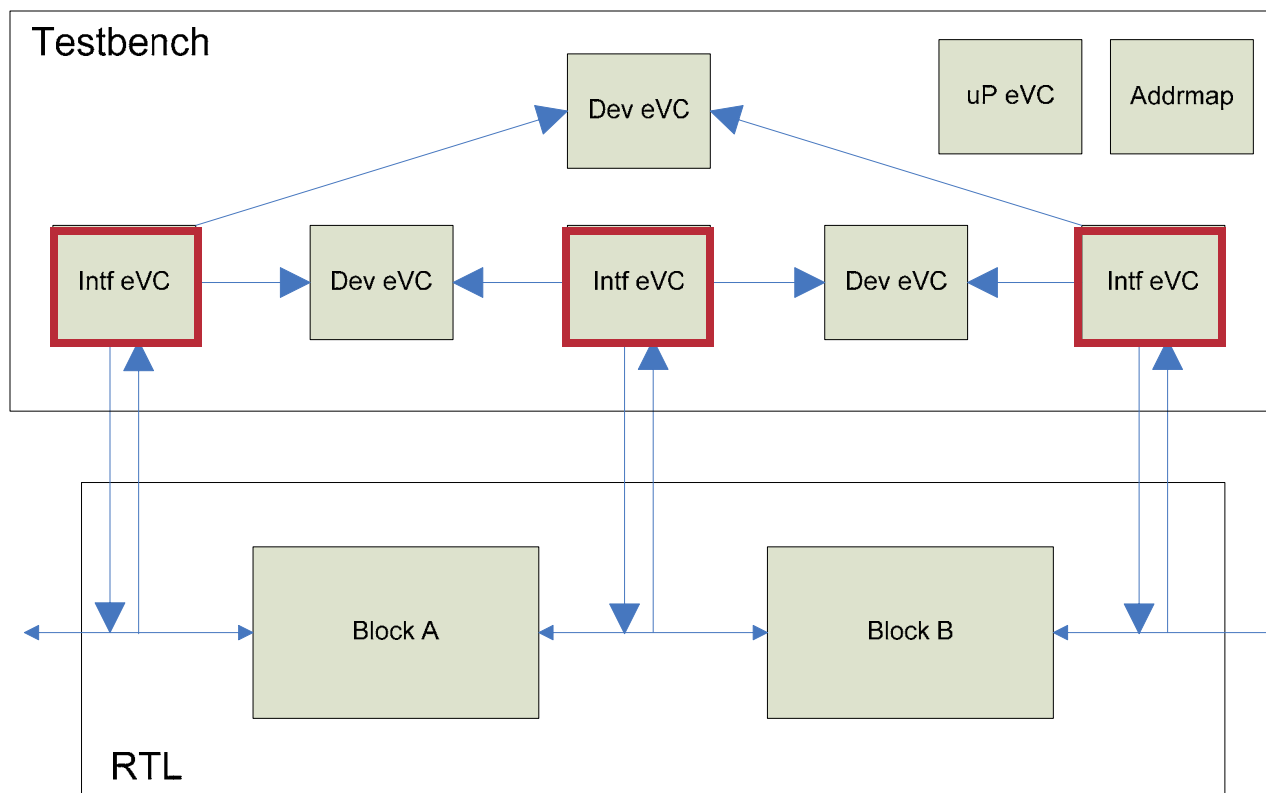


cscov_evc_topology Presentation

Mark Strickland

February 2006

Standard Testbench Infrastructure



eVC Characteristics

- Standard or custom
- Device, Interface or Virtual
- Present or not
- Active or Passive

cisco_evc_topology Package

Cisco.com

- **Template and script to create an interface eVC**
- **Automatically includes**
 - **Testflow mechanism (cisco_testflow)**
 - **Config structure (cisco_config)**
 - **Standard backpointer names**
 - **Comments for eDoc**
 - **Port for scoreboard hookup**
 - **Template for adding downstream drivers (e.g. backpressure, acknowledge)**

Standardization of Testcase

- **testphase sequence**
- **use <data>_seq or one_<data> in do's**
- **other constraints go in config**
- **evc_instance available for subtyping**

Usage Flow Overview

- ➔ **1. User selects name for eVC and data item and decides whether eVC requires subtyped agents**
- 2. Script produces files in a standard eRM package directory structure, checked into the Clearcase vob**
- 3. User adds behavior to some of these files to specify protocol details, etc.**
- 4. (Optional) If `cscovvc_topology` package needs to be updated, the script can be used to replace the core topology file without disturbing the rest of the completed eVC**

Name of eVC – *context_identifier_intf*

- ***context***

- For a standard interface, likely to be used by multiple designs, use **csc0**

- For an interface specific to one design, use either the design name or a 4-letter abbreviation (be consistent across all eVCs specific to that design)

- ***identifier***

- If interface has an obvious name based on the protocol or the block, use that name

- If interface does not have obvious name, use **block1_block2**, where **block1** and **block2** are connected to the interface

Need for Subtyped Agents

- **If interface has two or more distinct components that share much of the protocol but not all, subtyped agents are useful.**
- **Example: Transmit and Receive agents for communication interfaces**
- **With subtyped agents, the shared attributes go in the base agent and the differences go in the subtypes.**
- **Note: multiple agents that share all of the protocol are handled differently as will be seen later**

Usage Flow Overview

- 1. User selects name for eVC and data item and decides whether eVC requires subtyped agents**
- 2. Script produces files in a standard eRM package directory structure, checked into the Clearcase vob**
- 3. User adds behavior to some of these files to specify protocol details, etc.**
- 4. (Optional) If cisco_evc_topology package needs to be updated, the script can be used to replace the core topology file without disturbing the rest of the completed eVC**

Basic Script Usage

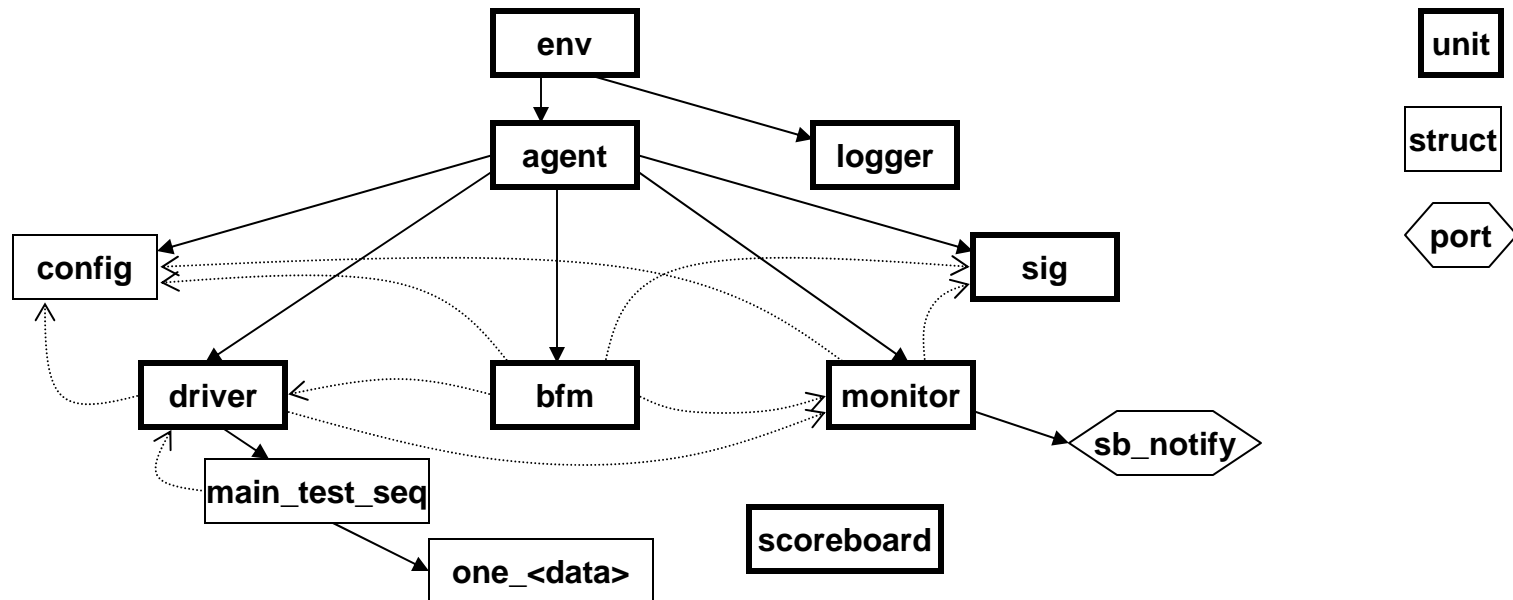
- **mkevc.pl <Required Args | -help> [Optional Args]**
- **Required args:**
 - **evc <evc_name> (see next slide)**
 - **data <data_name>**
- **One optional arg: (others discussed later)**
 - **agent <agent1[,agentX]>**

eVC Name and Vob Directory

- **Script enforces evc_name prefix and vob directory are consistent**
- `/vob/asicproc/verification/evc_lib`
 - interface eVC's for standard interfaces (cisco_xxx)
- `/vob/<project>/verification/evc_lib`
 - interface eVC's for internal or other non-standard interfaces (<proj>_xxx_intf)

Generic Topology (1 agent eVC)

Cisco.com



- names shown are field names, for both instances and back-references
- units have type names `<evc>_*_u`, where `<evc>` is eVC name and `*` is field name
- config struct is `<evc>_config_s`, sequence struct is `<evc>_<data>_seq`, data struct is `<evc>_<data>_s`
- all structs/units have `evc_instance` field, available for referring to multiple instances of this eVC as different subtypes (no need to worry about it if only one instance will exist)

Template for Downstream Driver

- **The interface may include some information that flows in the opposite direction to the data (e.g. backpressure, acknowledge).**
- **The script produces a template file that adds a driver to control these values to the agent, including all of the standard references.**
- **The eVC creator will copy this template to a file with an appropriate name and search/replace the names within to match (discussed in more detail later).**

Files Created in e Subdirectory

- Assuming chosen names are `my_evc` and `my_data`

`my_evc_intf_topology.e`

Describes all the infrastructure

`my_evc_intf_ports.e`
`my_evc_intf_config.e`
`my_evc_intf_my_data.e`
`my_evc_intf_monitor.e`
`my_evc_intf_bfm.e`
`my_evc_intf_my_data_seq.e`
`my_evc_intf_agent.e`
`my_evc_intf_env.e`

Empty shells for customizing extensions of each unit/struct

Available for integration into a device eVC

`my_evc_intf_scoreboard.e`

`my_evc_intf_downstream_template.e`

For copy/edit as needed

Usage Flow Overview

- 1. User selects name for eVC and data item and decides whether eVC requires subtyped agents**
- 2. Script produces files in a standard eRM package directory structure, checked into the Clearcase vob**
- 3. User adds behavior to some of these files to specify protocol details, etc.**
- 4. (Optional) If cisco_evc_topology package needs to be updated, the script can be used to replace the core topology file without disturbing the rest of the completed eVC**

Do Not Edit *_topology File

- **The intent is that the topology file could be regenerated by the script after the eVC is complete (in case transparent modification to the structure is needed).**
- **If the topology file has custom changes, these would be lost in a regeneration.**
- **Instead extend units/structs in other files**
 - In an extreme case, a unit/struct instance could be constrained to NULL**

evc_instance Field for Subtyping

- **All units have a field called `evc_instance`,**
 - enumerated type `<evc_name>_evc_instance_t`, initial range is `[NONE]`
- **The integrator can extend the enumerate type to add a value then use that value in the instantiation of this eVC**
- **Integration files can then extend the subtype of any unit that pertains only to a specific instance**

Add Port Info to Ports File

- Example

```
<'
package my_evc_intf;
import my_evc_intf_topology;

extend my_evc_intf_ports_u {
  -- bus CLOCK signal
  Clock      : in simple_port of bit is instance;
  keep Clock.hdl_path() != "" => bind(Clock,external) ;

  -- (active high) Write signal
  Write      : inout simple_port of bit is instance;
  keep Write.hdl_path() != "" => bind(Write,external) ;

  -- 19 bit Read Address
  RdAddr     : inout simple_port of uint(bits:19) is instance;
  keep RdAddr.hdl_path() != "" => bind(RdAddr,external) ;

};
'>
```

A signal such as clock that should never be driven by the eVC can be declared with direction in. Every signal that can be driven should be declared with direction inout, not out, as we do not want to inhibit sampling of it later on.

Use "uint(bits:)" for wide signals

Config Built-In Fields

- **config_mode**
- **use_sequence : bool**
- **use_checks : bool**
- **upstream_active : bool**
- **downstream_active : bool**
- **active_passive**

Using Config Fields

- **All configuration fields should exist in the config struct and, in general, not in the other units**
- **If a unit needs to be subtyped based on a configuration field**
 - include the field in both the config struct and the required unit
 - use constraint of this form in the unit
`keep <field_name> == read_only(config.<field_name>)`

Data Item Built-Ins

- like `cscotrans_s`
- kind : [GENERIC, DRIVER, MONITOR]

Data Item Customization

- **Add fields, methods, and constraints as usual for describing the data item.**
- **If you want to base the data item on `cscotrans` and get the associated transaction tracking benefits, define `<EVC_NAME>_USE_CSCOTRANS` in the top e file before all the imports.**

Clock Ports

- **All units except the unit called ports contain a port called `clock_edge`.**
 - Allows `@clock_edge$` as standard sampling event
- **By default, all the clock ports within the eVC are bound together in the `connect_ports()` phase**
 - Could extend `connect_ports()` to unbind as needed
- **Expected use is either the monitor clock port is emitted based on an external signal or the env clock port is bound to a clock from a higher level**

Testflow

- env, monitor, bfm, driver use testflow
- **Built-in methods for each phase**

```
•do_pre_reset() @clock_edge$ is {};  
do_reset()      @clock_edge$ is {};  
do_init_test()  @clock_edge$ is {};  
do_main_test()  @clock_edge$ is {};  
do_post_test()  @clock_edge$ is {};  
do_post_test_check() @clock_edge$ is {};
```

- **All units must complete phase before any units start the next phase**
- **Driver does not complete phase until any started sequences complete**

Monitor Built-Ins

- **rx_<data>**
- **sb_notify**

Example Monitor

```
<'
package my_evc_intf;
import my_evc_intf_topology ;

extend my_evc_intf_monitor_u {

    !saved_data : list (key : addr) of spyg_pm_intf_monitor_meta_data_s;

    -- Write event
    event wr_event ;
    -- Read event
    event rd_event;

    event clock_rise is rise(sig.Clock$)@sim ;
    on clock_rise { emit clock_edge$ };

    on wr_event { sb_notify$(rx_word) };

    main_body() @clock_edge$ is also {
        start write_body() ;
        start read_body() ;
    };
};
'>
```

The monitor may save history of received values. (Note meta_data_s is not a built-in and would be declared elsewhere in the full example.)

Create an event for every meaningful signal change. The other units will know about DUT state only through these events. The comments will show in eDoc.

The monitor must declare the clock event and emit the clock event port. It should use the event port internally for consistency.

The monitor must call the method port sb_notify at the time when data is ready for the scoreboard

So that one can consistently find the full description of what happens in the main test phase, any parallel threads that need to be launched throughout the phase should be started from main_body(). Since main_body() does not consume time, monitor will not prevent main_test phase from finishing.

Example Monitor Continued

```
<'
extend my_evt_intf_monitor_u {

write_body() @clock_edge$ is {
  while TRUE {
    message(HIGH,"waiting for write = 1");
    wait true(sig.WrAddr$ == 1) ; -- signals valid data is present
    rx_word = new ;
    rx_word.addr = sig.WrAddr$;
    rx_word.data = sig.WrData$;
    rx_word.meta = sig.WrMeta$;
    emit wr_event ;
  };
};

on wr_event {};

extend has_checks my_evt_intf_monitor_u {
  expect wr_event => not wr_event; -- no back-to-back writes
};
'>
```

Monitor process will be "while TRUE" as the the sequence driver will determine the end of the phase.

rx_<data> (word in this case) is a built-in field to hold the received data.

Emit the events at the appropriate place in the protocol.

Could add to saved history here

Put all protocol checks in the has_checks subtype.

BFM Built-Ins

- **upstream_active subtype**
 - extend with process sending primary data
- **downstream_active subtype**
 - extend with process sending responses (if any)
- **get_next_<data>()**
 - returns data from the driver if use_sequence is TRUE (the default)
- **reset_bus()**
 - a method to be extended to set all driven values to their quiescent condition

BFM Customization

```
<
package my_evc_intf;
import my_evc_intf_topology;

extend upstream_active my_evc_intf_bfm_u {
  do_main_test() @clock_edge$ is also {start main_body()};
  main_body() @clock_edge$ is also {
    var wrd : my_evc_intf_word s;
    while TRUE {
      wrd = get_next_word();
      message(MEDIUM,"sending word ",wrd);

      send_a_word(wrd);

      message(MEDIUM,"finished sending word ",wrd);
      if use_sequence {emit driver.item_done};
    };
  };

  reset_bus() is also {
    sig.RdMeta$ = 0x0;
  };
};
'>
```

BFM process will be "while TRUE" as the the sequence driver will determine the end of the phase.

Use get_next_<data>() to return next data item

send_a_word() would have to be defined elsewhere in the example.

Emit item_done conditioned by use_sequence boolean.

reset_bus() is a built-in method to be extended with any assignments the eVC should make in the reset phase.

Driver/Sequence Built-Ins

- **Sequence for each testflow phase, automatically started in that phase**
 - sequence subtypes : PRE_RESET, RESET, INIT_TEST, PRE_TEST, MAIN_TEST, POST_TEST
 - Example: to specify the xyz_intf_packet sequence behavior for the init_test phase, extend INIT_TEST xyz_intf_packet_seq
- **Sequence field <data>_seq, data item field one_<data>**
 - body() can “do XXX packet_seq” or “do one_packet”

Driver/Sequence Customization

- **Add any constraints that are required across all sequences.**
- **Add any generic sequences**
 - **Example: READ and WRITE sequences for a transaction interface**

List of Drivers

- **If the interface drives a single data stream which represents the interleaving of multiple data streams, then a list of drivers should be used.**
 - Each driver represents one of the interleaved streams
 - The bfm will get data from each driver and perform the interleaving
- **To enable a list of drivers in the topology file, define <EVC_NAME>_DRIVER_LIST in the top file before the import of the topology file.**

List of Agents

- **If the interface drives a multiple data streams (with identical protocol), each through its own port, then a list of agents should be used.**
 - Each agent represents one of the data streams
 - An additional field, `agent_idx:uint`, is propagated through the structure to identify the parent agent's position in the list
- **To enable a list of agents in the topology file, define `<EVC_NAME>_AGENT_LIST` in the top file before the import of the topology file.**

Agent Customization

- **If a list of drivers is used, the agent will contain the constraint on the list size.**
- **Otherwise, the agent will typically be left empty, but anything that should be at the level of the monitor/bfm can be added.**

Env Customization

- **If a list of agents is used, the env will contain the constraint on the list size.**
- **Otherwise, the env will typically be left empty, but anything that should be at the level of the agent(s) can be added.**
 - Example: a locker representing a resource that must be shared by all the agents**

Downstream Driver Customization

- **To control downstream responses**
 - copy the *_downstream_driver.e file to your desired name (recommendation: <evc_name>_<type>_control_driver.e
 - open the new file in a text editor and substitute your desired name (recommendation: <type>_control) for XXXdownstreamXXX
 - add the new file to the import list in the top file, after the topology file but before the bfm file

Downstream Driver Built-Ins

- **Data item**
 - duration
 - latency
- **Same sequence hooks**
- **BFM gains `get_next_<name>()` method**

BFM Downstream Customization

```
<'
package my_evc_intf;
import my_evc_intf_topology;

extend downstream_active my_evc_intf_bfm_u {
  do_main_test() @clock_edge$ is also {start main_bp()};
  main_bp() @clock_edge$ is also {
    while TRUE {
      var bp_control : my_evc_intf_bp_control_s;
      sig.Bp$ = 0;
      bp_control = get_next_bp_control();
      wait [bp_control.latency];
      sig.Bp$ = 1 ;
      wait [bp_control.duration] ;
    }
  };
};
'>
```

This method is also an endless loop, terminated when the testflow completes

Use get_next_<data>() to return next data item

Downstream data provides latency and duration fields.

Top File Customization

- **Add defines for multiple drivers, multiple agents, and/or cisco_trans use.**
- **Add imports for any files that were manually created.**
- **May need to use cyclic import or header file strategy if code additions in multiple files depend on each other**

Usage Flow Overview

- 1. User selects name for eVC and data item and decides whether eVC requires subtyped agents**
- 2. Script produces files in a standard eRM package directory structure, checked into the Clearcase vob**
- 3. User adds behavior to some of these files to specify protocol details, etc.**
- 4. (Optional) If cisco_evc_topology package needs to be updated, the script can be used to replace the core topology file without disturbing the rest of the completed eVC**

Topology Update

- **Reminder**
mkevc.pl <Required Args | -help> [Optional Args]
- **Another optional arg:**
 - **update_topology**
- **Replaces topology file in chosen eVC**
- **Allows package developer to update topology template and push it out to all existing interface eVCs (of course, not every change possible will be transparent)**

Examples of Resulting eVC's

- In **/vob/newby/verification/evc_lib/**
 - newb_wrfifo_intf
 - uses multiple drivers
 - newb_rdfifo_intf
 - uses multiple agents
 - uses two downstream drivers: backpressure and ack
- In **/vob/asicproc/verification/evc_lib/**
 - cscs_csr_master_intf
 - locker in env synchronizes agent behavior

CISCO SYSTEMS



EMPOWERING THE
INTERNET GENERATION