Migrating existing AVM and URM Testbenches to OVM

Paradigm Works

Steve D'Onofrio
Ning Guo

Table of Contents

1 ABSTRACT 3

2 TESTBENCH ARCHITECTURES 4

2.1 PW Router Design 4

2.2 AVM Testbench 5

2.3 URM Testbench 7

2.4 OVM Testbench 9

2.5 AVM/URM to OVM Code Migration 12

3 CONFIGURATION CONTROL 14

4 OVM SEQUENCE MECHANISM 17

4.1 Sequence 17

4.2 Sequencer 19

4.3 Driver 20

4.4 Virtual Sequencer and Virtual Sequence 21

5 THE OVM FACTORY - PARITY ERROR EXAMPLE 23

6 AUTOMATING OVM TESTBENCH GENERATION 25

7 CONCLUSION 26

1 Abstract

OVM (Open Verification Methodology) is the result of joint development by Cadence and Mentor Graphics. It combines the Cadence incisive Plan-to-Closure Universal Reuse Methodology (URM) and the Mentor Advanced Verification Methodology (AVM).

As users of both AVM and URM methodologies, we have existing testbenches that were developed for each individual methodology. During the process of migrating from our existing AVM-only and URM-only testbench to OVM testbench, we were able to understand better on how the two methodologies complement each other in the OVM.

Using the same design under verification, we will describe the testbench facilities in each methodology.We will specifically discuss aspects of stimulus generation, response checking, scoreboarding, and testbench architecture. We will then briefly describe our OVM testbench's configuration control mechanism, virtual sequence, and factory capabilities.

Finally, we will talk about generating an OVM based testbench automatically using a template generator. The template generator allows users to generate a customized OVM-based environment.It enforces a consistent look and feel, and enables rapid development and maintenance of the verification code across multiple-sites and cultural barriers.

2 Testbench architectures

This section describes the PW Router design and gives a high-level overview of the URM, AVM, and OVM testbench architectures that we put together to verify the PW Router design.

2.1 PW Router Design

At Paradigm Works, we developed a plethora of testbenches against an in-house Design-Under-Verification (DUV) - the "PW Router". The PW Router has a single input packet interface and three output packet interfaces. It also has a host interface which allows access to the status and control registers in the design. Figure 1 shows the block diagram of the device.

The PW Router design has many features. Among them is parity checking. Parity check verifies the packet contents entering and exiting the design. The host processor may configure the design to use either odd or even parity checking. In addition, the host may configure the design to drop packets that fail the parity check.

Other features that are beyond the scope of this paper include:

- Interrupt controller
- Packet length check engine
- Forwarding engine that directs packets to the output ports based on the address and priority fields in the packet. The router intermediately stores packets to one of the four external memories that are represented by the router's priority.
- The PW Router may be configured using fixed or round robin arbitration schemes.
- Packet FIFO underflow and overflow error detection logic
- Packet segmentation engine

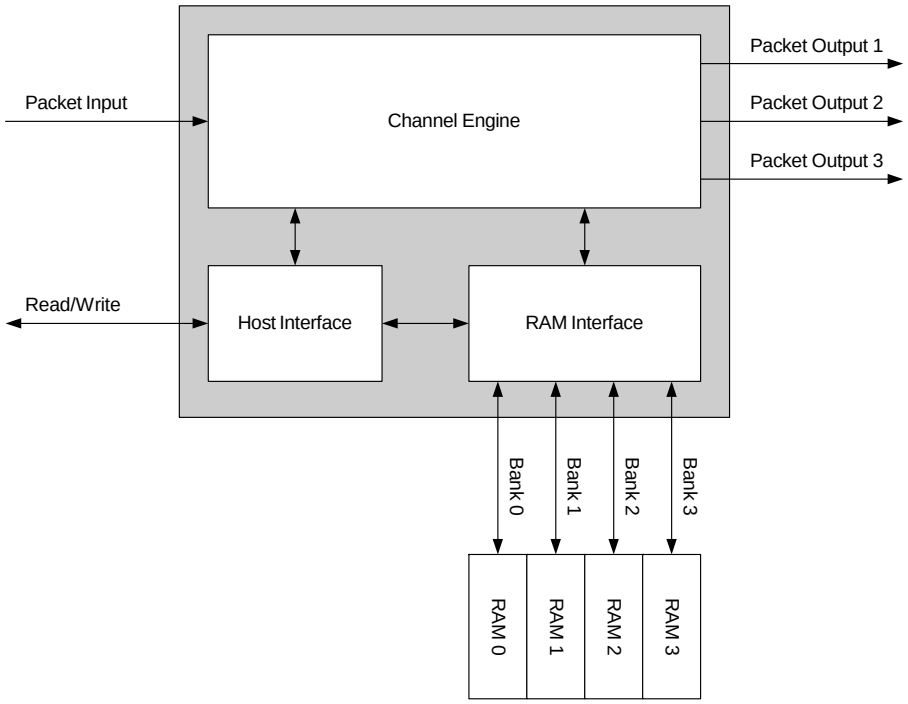

Figure 1 PW Router DUV

2.2 AVM Testbench

Figure 2 shows the AVM Testbench that we put together for the PW Router DUV. The AVM Cookbook states that an AVM Testbench is broken-up into two domains; the operational domain and the analysis domain. The operational domain is the set of components, including the DUV, that operate the DUV. This includes the stimulus generators, BFMs and similar components, the DUV, responder and driver — along with the environment components that directly feed or respond to drivers and responders. The rest of the testbench components, i.e., monitors, scoreboards, coverage collectors and controller, comprise the analysis domain. These are the components that collect information from the operational domain.[i]

One large way AVM promotes component reuse is by using TLM (Transaction-Level Modeling). TLM allows the testbench components to communicate with each other by sending transactions back and forth through channels. Using transactions eliminates the need for referencing testbench-specific components (pointers) within other components that diminished reuse. TLM is based on the OSCI Standard. The abstraction level of the transaction may vary at the product-description rather than the physical-wire level. This allows components to easily be swapped in and out without affecting the rest of the environment.

A special TLM port, the analysis port, forms the boundary between the operational domain and the analysis domain in an AVM testbench. The analysis domain is responsible for analyzing the behaviors observed by the monitors in the testbench. A monitor converts the operational domain's bus activity into higher-level abstraction transactions. The monitor component (publisher) broadcasts transactions in

zero-time (non-blocking) to one or more analysis domain components (subscribers). The subscribers are able to store an unbounded amount of transaction in its analysis fifo in the case if the subscriber operates slower than the publisher.

The AVM Cookbook points out that the analysis domain components answer two questions:

- Does it work?
- Are we done?

Scoreboard components are recommended to determine if the design is working. Scoreboards collect a list of expected transfers (perhaps through the use of a transfer function) and compare them against the actual DUV response. Monitor components may also use SystemVerilog Assertions (SVA) to verify the correctness of the DUV. In general, analysis domain components send error status to the test controller. The test controller is responsible for taking appropriate testbench action based on its configuration and the error condition.

Coverage components contain SystemVerilog functional coverage constructs and can answer the “Are we done” question. The AVM Cookbook also recommends feeding back information based on coverage data into the test controller for reactive style testing, but it is our experience that reactive testing does not scale well with complex SoC designs.

Figure 2 shows the two domains and the TLM ports.

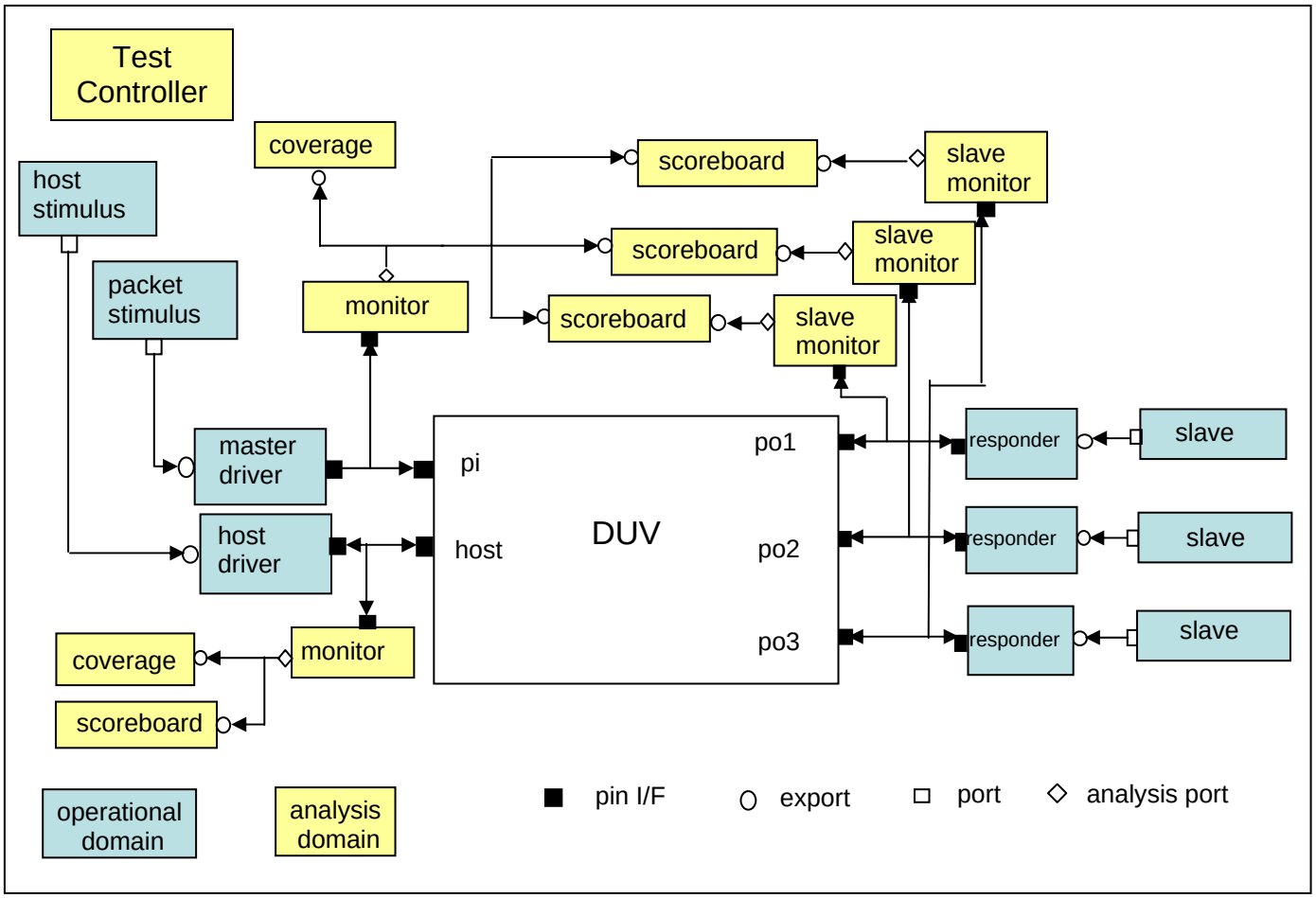

Figure 2 AVM Testbench Architecture

2.3 URM Testbench

Figure 3 shows the URM testbench architecture that we put together for the PW Router DUV. The URM Testbench architecture is layered and highly configurable. The URM configuration mechanism allows fields in a verification component to be configured at various layers within the testbench. In addition, the URM includes factory capabilities. All these concepts are described in detail later in this paper.

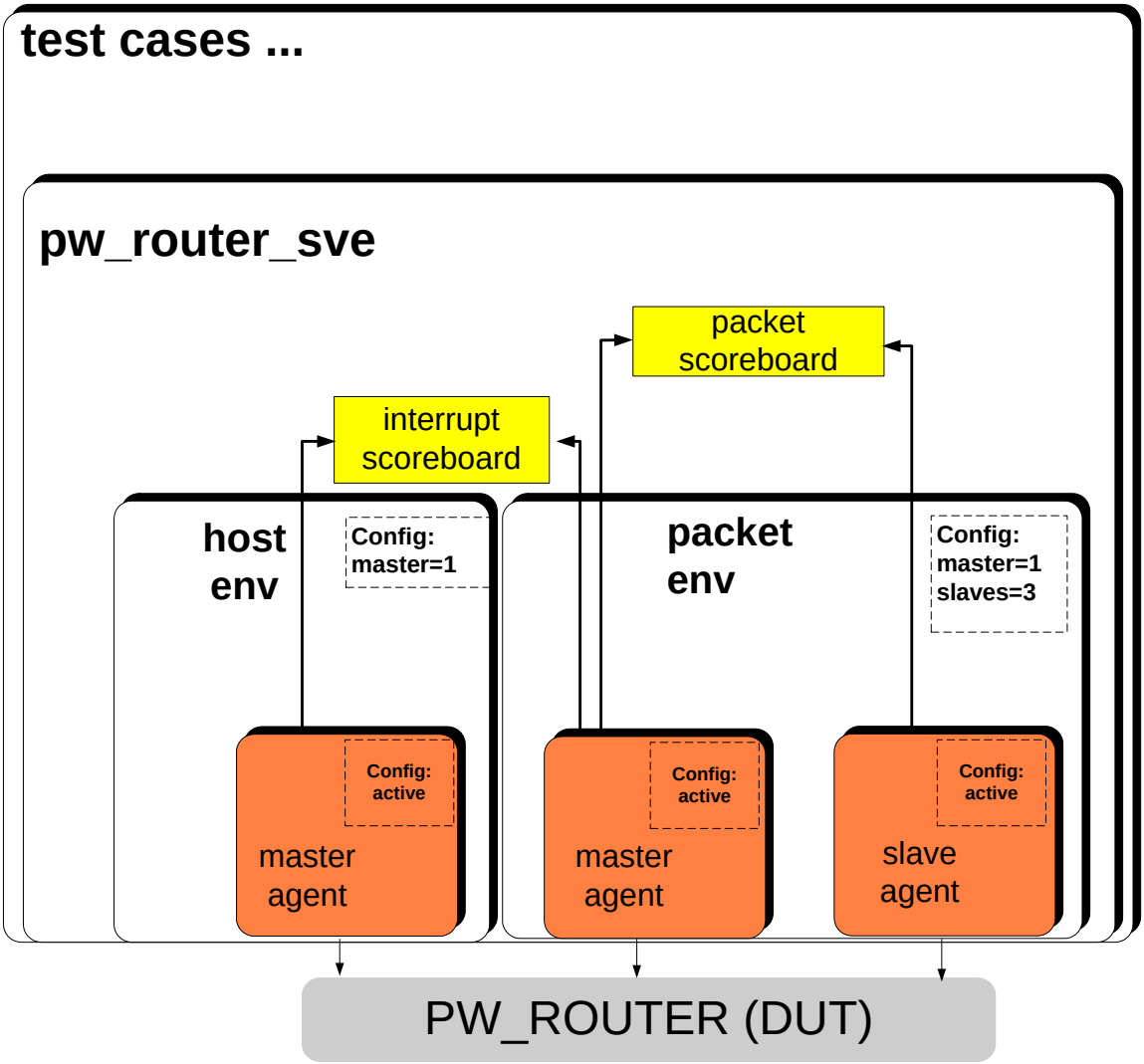

Figure 3 URM Testbench Architecture

An Interface UVC is a component specific to a particular interface protocol. It consists of one or more agents. An agent is a component that connects to a single port on the DUV. An agent can be configured as either active or passive. Figure 4 shows the block diagram of an agent. An active agent is made up of a BFM, a driver and a monitor. It drives stimulus into the DUV and captures the bus activities of the DUV. A passive agent includes only a monitor.It captures bus activity but does not drive stimulus into the DUV.

PW Router DUV has one packet input port and three packet output ports. In our PW Router URM testbench, we developed an interface UVC, *packet_env*, to communicate with the packet interface. The packet_env includes one master agent and three slave agents (Figure 3). The master agent is connected to the packet input port and the slave agents are connected to the packet output ports. We developed another interface UVC, *host_env*, to drive and monitor host traffic. All the agents inside the packet_env and host_env are configured in active mode.

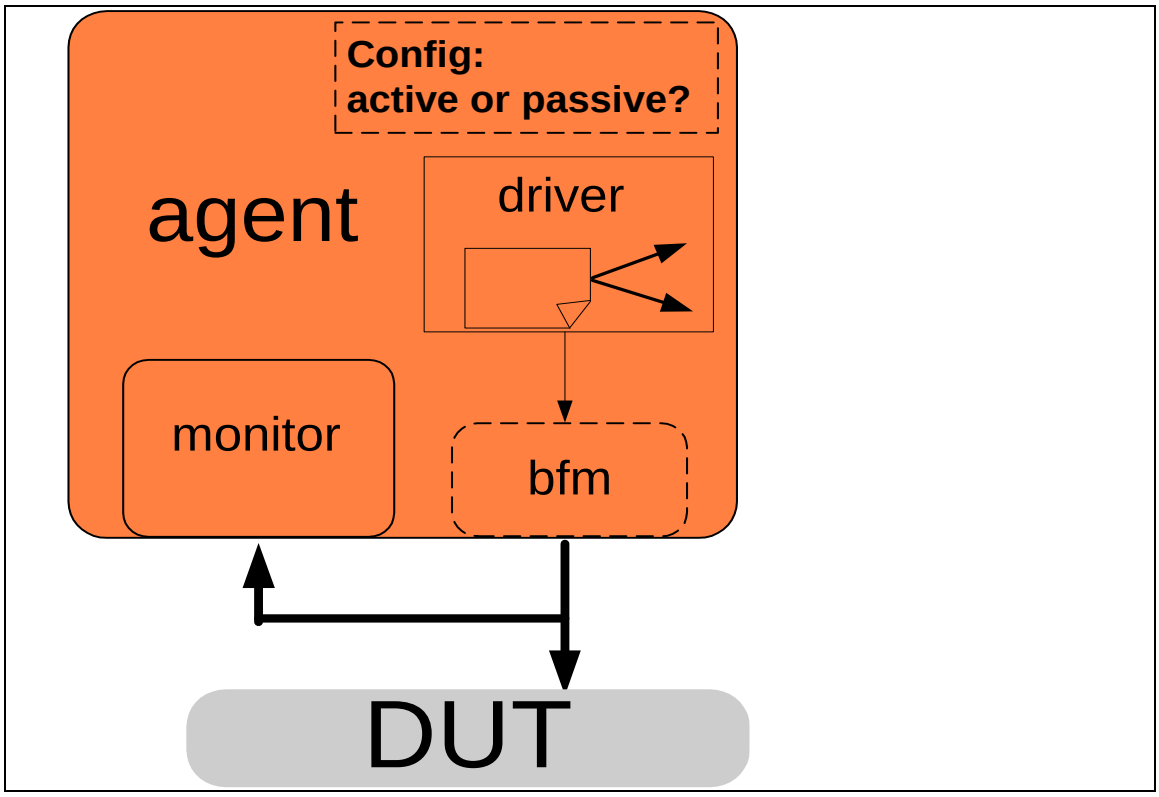

Figure 4 Agent

Figure 3 also showed that the URM testench for PW Router includes two container layers that are specific to the testbench (i.e. they are not intended for reuse): the pw_router_sve and test cases.

The pw_router_sve encapsulates the interface UVCs. It may encapsulate sophisticated module UVC(s) and virtual sequences but we opted to exclude those components in this version of URM testbench due to time constraints. The testcase layer allows users (sometimes referred to as test writers) to customize testbench controls.

2.4 OVM Testbench

OVM combines concepts from both AVM and URM. It is backward compatible with AVM and URM codes. There are some minor nomenclature changes for OVM as shown in Table 1. The remainder of this paper will refer to the OVM naming convention.

URM name	OVM name
BFM	Driver
Driver	Sequencers

Table 1 Differences in URM and OVM names

Since URM and AVM have significantly different methodology styles but OVM could support both styles, users must make a decision on what path they intend on following in terms of the aspects listed in Table 2.

	AVM	URM	OVM
Testbench structure	Layered - Test Controller - Operational Domain - Analysis Domain	Layered -Test cases -SVE -UVCs	**Support both**
Component structure	Flat/ TLM Channels	Layered/ Structured / Configurable	**Support both**
Stimulus	avm_random_stimulus	Sequences	**Support both**
Configuration/ Factory	No AVM library features	Configuration/ URM Factory	**Configuration/ OVM Factory**
Scoreboard/Coverage	Analysis port/TLM	No guidelines	**Analysis port/TLM**

Table 2 AVM vs. URM & what is supported by OVM

A key aspect of developing efficient reusable verification code is to design a testbench architecture that is made up of multiple layers of highly configurable components. URM provides components that are reusable from protocol level of abstraction (interface UVCs) and module level of abstraction (module UVCs). Interface and module UVCs coupled with the URM configuration/factory mechanism provides all the hooks needed to reuse components from testbench to testbench. Therefore, we decided to go with URM style testbench architecture for PW Router OVM testbench.

We also decided to use TLM channels for connections between interface OVCs(OVM Verification Component) and the scoreboards and coverage components. This allowed us to easily reuse the scoreboard and coverage components from our AVM testbench. We did not use TLM channels between components inside packet and host interface OVCs because these interface OVCs are single entity that will not be taken apart. However, this may not always be the case for all OVCs.

For stimulus generation we opted to use the sequence mechanism in our OVM testbench. Section 4 will talk about the details of this process.

Figure 5 illustrates the OVM testbench that we put together for our PW Router DUV. The packet and host interface OVCs are reused from our URM testbench, the scoreboard components are reused from our AVM Testbench. In addition, we added a virtual sequence (pwr virtual sequence) and module OVC (pw_router_env). The OVM testbench also has two container layers that are specific to this testbench: the pw_router_sve and test cases.

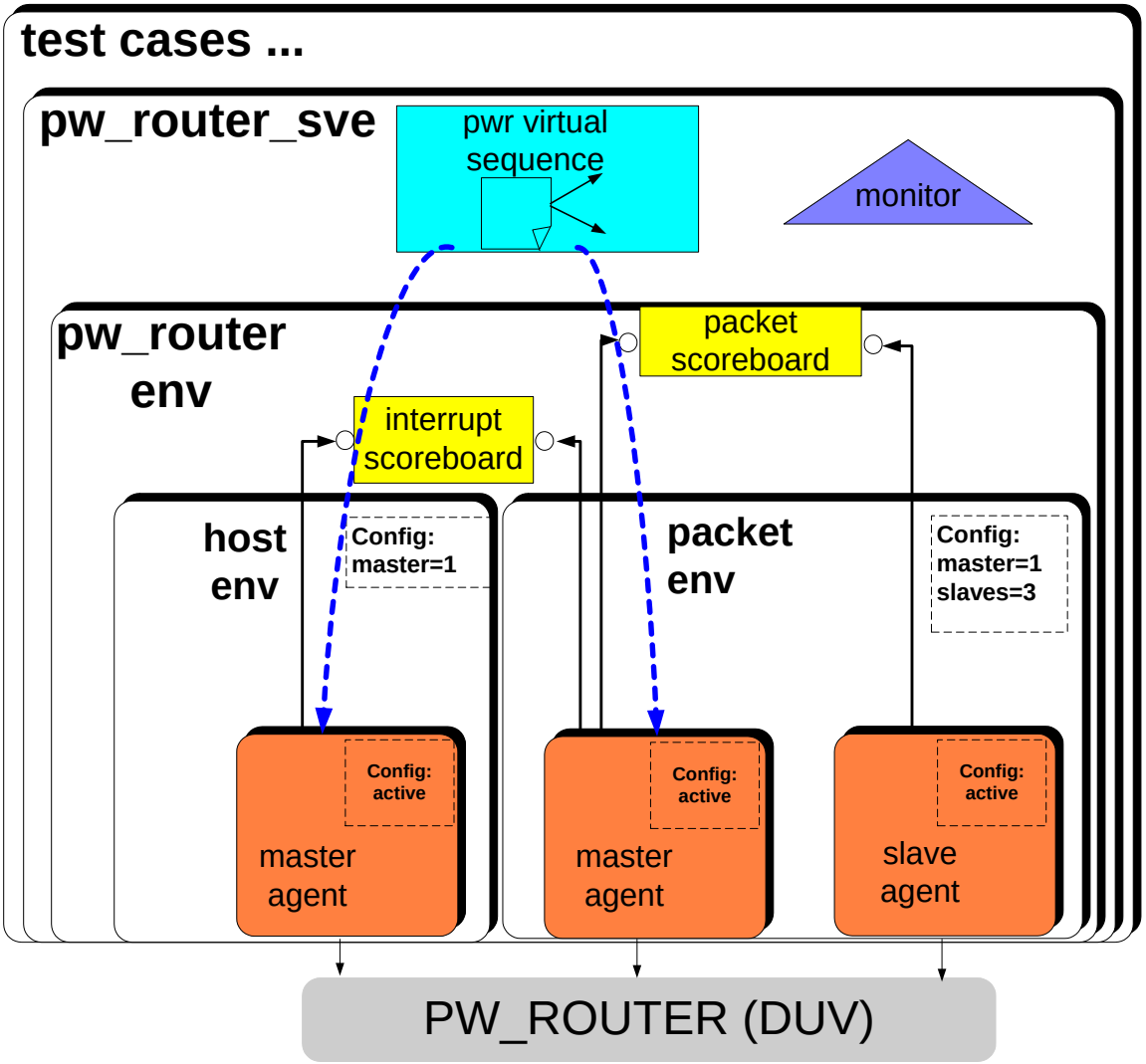

Figure 5 OVM Testbench Architecture

Virtual Sequence coordinates activities among multiple OVCs and will be discussed in section 4. *Module OVC* promotes reuse at the module level of abstraction. It can be used in multiple testbenches, such as the unit-level testbench and system-level testbench. For example, Figure 6 shows a system-level testbench for a PW_TOP design that we are planning to implement in the future. This design encapsulates the PW Router design and a PW AHB design. The system testbench contains the two module OVCs, i.e., pw_router_env and pw_ahb_env.

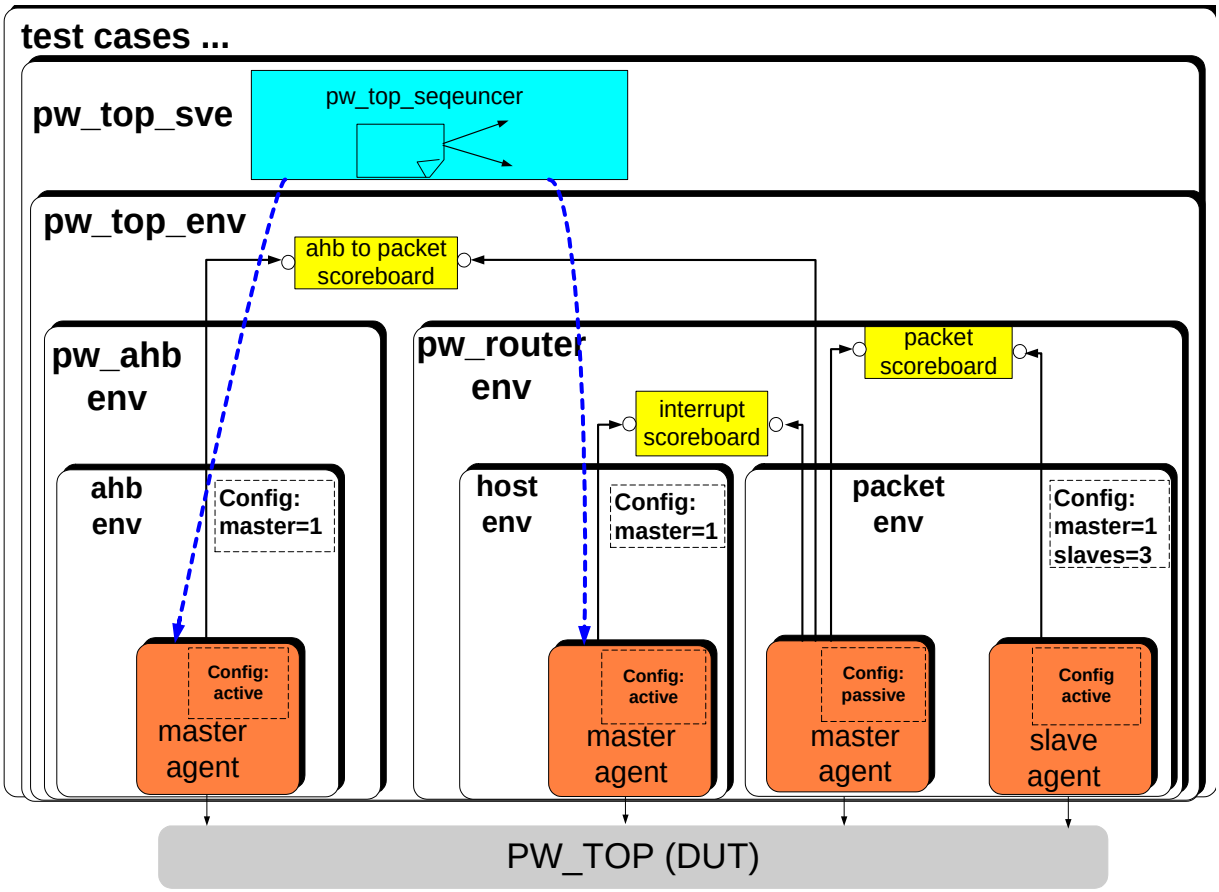

Figure 6 Future OVM System Level Testbench

One thing needs to mention is that in the system-level testbench, the master agent of the packet_env in pw_router_env needs to be configured as a passive agent. This is because the packet interface is now driven by the PW AHB device and hence the testbench can only monitor activities on this interface. However, putting the agent in passive mode does not affect the scoreboards connected to it in the pw_router_env because scoreboards obtain data from the monitor anyway.

2.5 AVM/URM to OVM Code Migration

The OVM library has AVM and URM compatibility facilities. Our PW Router AVM 2.0 testbench was able to run with OVM library without any issues. Most of the URM classes, macros, and messages were able to migrate from URM to OVM without any issues. However, we did experience several minor annoyances as shown in the list below and in Figure 7.

- The OVM macro utility fields have “OVM_” in their name
- OVM introduced a new macro for enumerators
- The component’s new constructor in OVM has a different prototype argument names

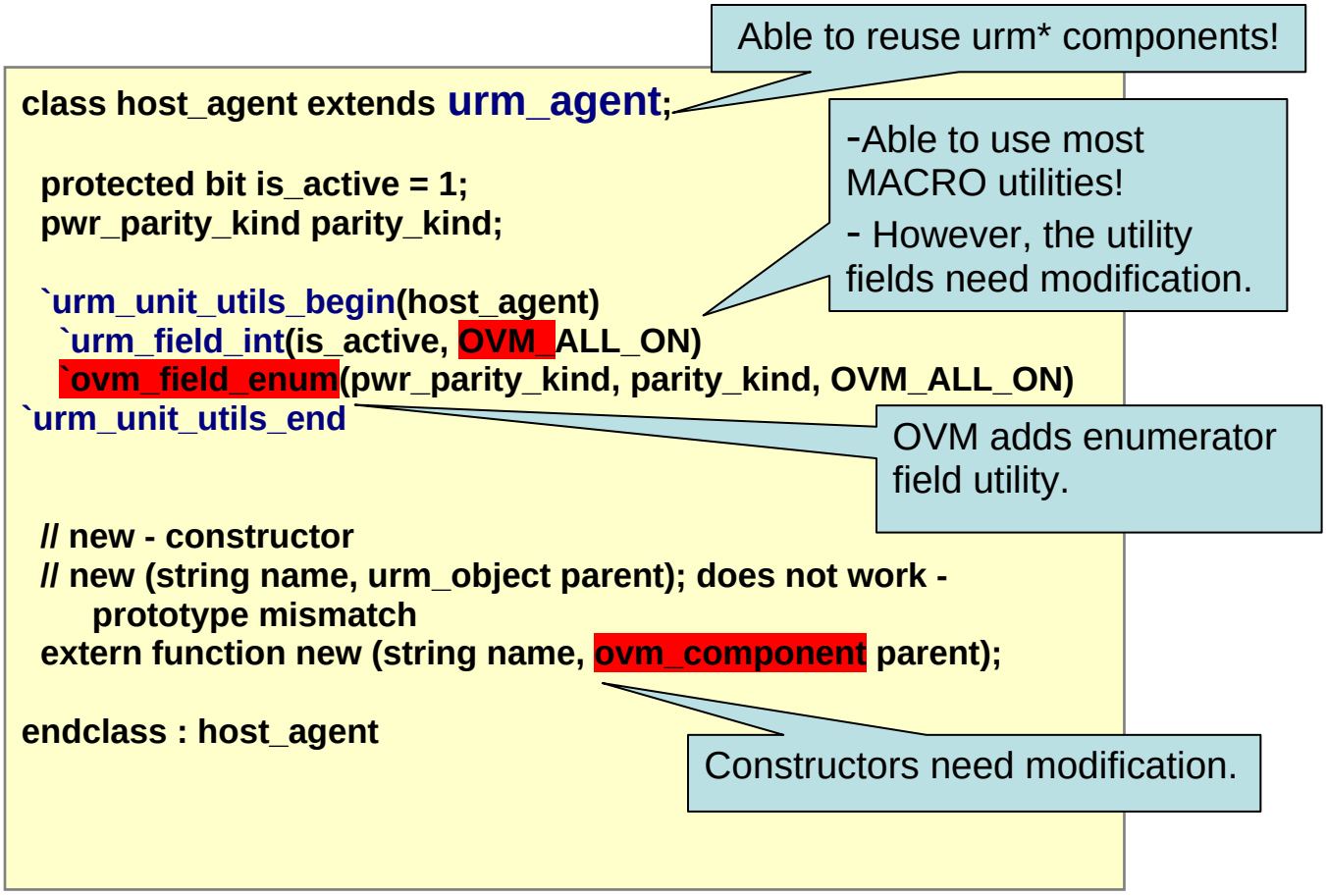

Figure 7 URM to OVM Code Migartion

3 Configuration control

OVM components are self-contained. The behavior and implementation are independent of the overall testbench, facilitating component reuse. The components are built using recursive construction, that is, the parent component builds only its immediate children. Children components in turn then build their own immediate children components[ii].

Typically, components have fields (sometimes referred to as 'knobs') to control their modes of operation. There are two flavors of fields:

- Topology configuration fields such as the active_passive field of an agent
- Behavior configuration fields that may control testbench activities such as the ODD or EVEN parity mode

OVM provides advanced configuration capabilities. The primary purpose of configuration mechanism is to control the field value during the build phase. The build phase occurs before any simulation time is advanced. The fields may also be changed during simulation time (or the run phase) but this capability is beyond the scope of this paper.

The configuration mechanism gives test writers and higher layer testbench components (i.e. module/system OVCs) the ability to overwrite the default field settings of the components. Figure 8 shows the testbench layers from right (lower layer) to left (higher layer). The components on the left can overwrite the configuration set by the components on the right.

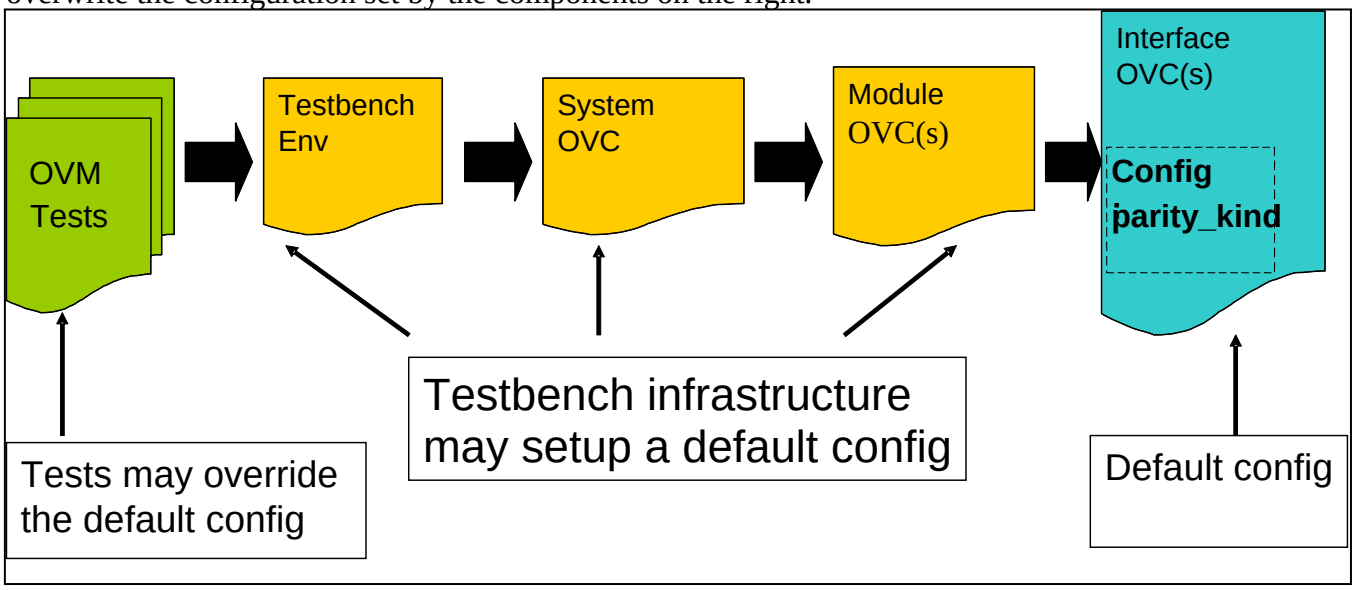

Figure 8 Testbench Configuration Flow

Figure 9 and 10 present code examples of configuration control in PW Router testbench. Recall that the PW Router DUV can operate with either ODD or EVEN parity. In the testbench, the host interface OVC needs to know the parity_kind to program configuration register correctly, the packet interface OVC needs to know the parity_kind to generate packets properly. Figure 9 shows the "parity_kind" field

declaration in pi_master_sequencer class and host_sequencer class. The OVM macro registers the field with the factory for it to participate the configuration mechanism.

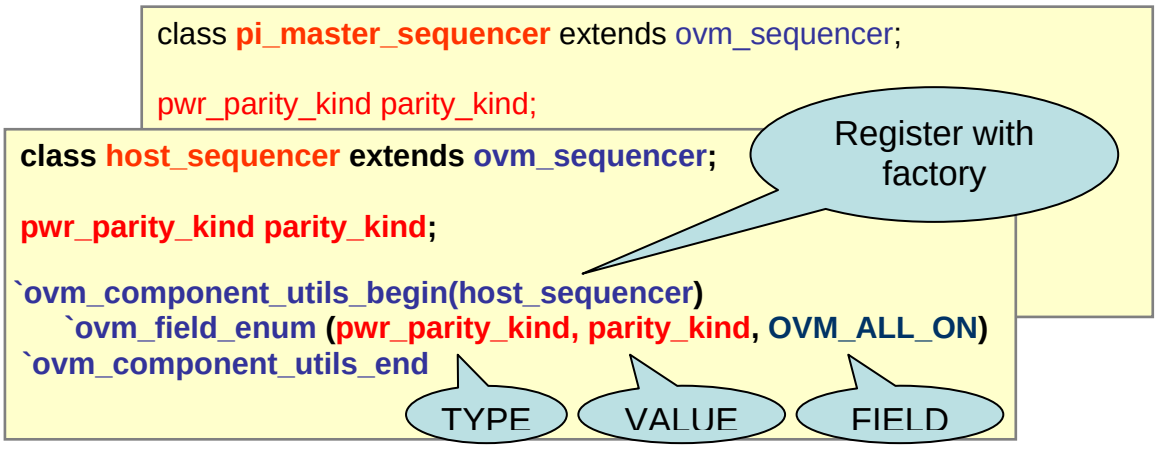

Figure 9 Interface uVC's parity config field

Since the parity_kind field of the host_sequencer and pi_master_sequencer are independent of each other, we need to synchronize them from higher layer of the testbench. In Figure 10, we declared and registered a parity_kind field in the pw_router module OVC. The module OVC's parity_kind field contains the default value used by the OVM PW Router Testbench. We declare the parity_kind field using the SystemVerilog 'rand' keyword so that it defaults to run with a random value

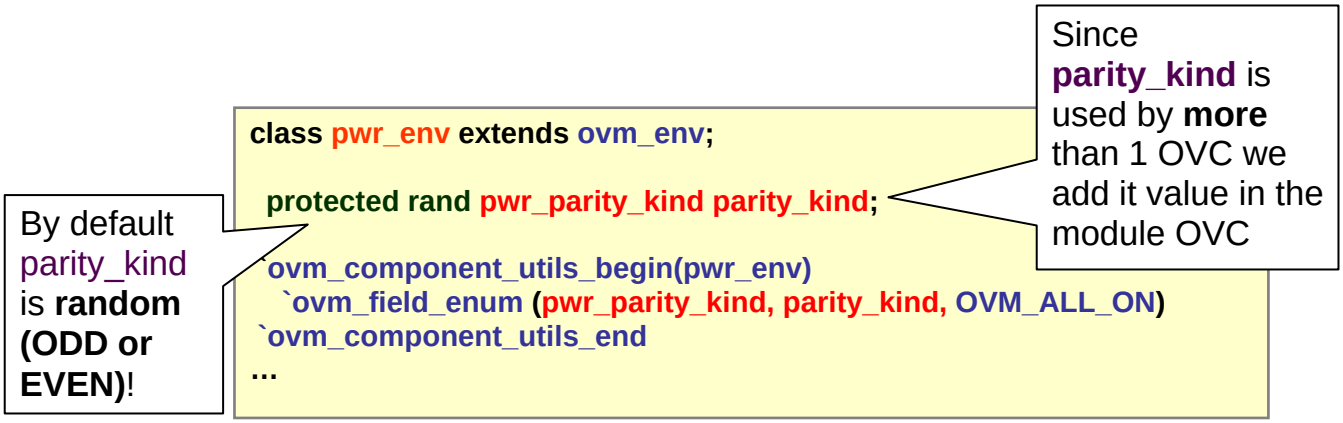

Figure 10 Module UVC's parity config field

Figure 11 shows a snippet of the OVM sve hierarchy and build() method inside the pwr module OVC. The sve is built when calling the super.build(). The pwr module OVC instance pwr0 is created using the create_component() OVM method. Users may explicitly call the build() OVM method after creating the component. However, we opted to let OVM implicitly handle building the pwr0. Randomizing the pwr0 must be called after creating it.

```
class pwr_sve_env extends ovm_env;

virtual function void build();
   super.build();

  $cast(pwr0, create_component("pwr_env", "pwr0"));
   assert(pwr0.randomize());

…
 endfunction : build
```

Randomize after creating pwr0 component

Figure 11 Randomize Module UVC fields

The next step is to push down the parity_kind field in the module UVC to the lower layer interface UVCs. This occurs in the module UVC's build phase as shown in Figure 12. The set_config_int OVM method is used to push down the value of the parity_kind of module UVC. The first argument in set_config_int() call is a wildcard '*', it performs a top-down search through all lower layer components (includes the host and packet interface UVCs) looking for a match on field "parity_kind". When a match is found, the "parity_kind" in lower layer components are assigned the value of the "parity_kind" of the module OVC. It is vital for the set_config_int OVM method to be called before the interface OVC's create_component() method. It is also very helpful to call the interface OVC's print() method to check if the fields are pushed down as expected.

```
class pwr_env extends ovm_env;

   virtual function void build();
     super.build();

     set_config_int(      "*“     , "parity_kind", parity_kind );


       // host env sub-component
     $cast(host0, create_component("host_env", "host0"));


      // pi env sub-component
      $cast(pi0, create_component(“packet_env", “pi0"));
```

set_config*
- Searches top-down
-May use wildcards

Note: make sure to call set_config* **before** creating the agents

Hint: for debugging set_config_* call host0.print() and pi0.print() here

Figure 12 Module OVC pushing party field to interface OVCs

Testcase layer can overwrite testbench parity_kind as shown in Figure 13. The “test_odd_parity_kind” testcase forces the parity_kind in the module OVC to always be ODD. Note that the set_config_* method is used to control the field setting and it must be called before calling super.build() which ultimately builds the testbench.

```
class test_odd_parity_kind extends pwr_base_test;
`ovm_component_utils(test_parity_kind)

virtual function void build();

    set_config_int("pwr_sve0.pwr0", "parity_kind", ODD);

  super.build();
 endfunction : build

endclass : test_odd_packet_parity
```

Figure 13 Test override default parity

4 OVM Sequence Mechanism

OVM sequences allow test writers to control and generate stimuli to the DUV. The sequence mechanism may be flat, layered, hierarchical (sometimes referred to as nested) layered, and controlled from higher layers of abstraction using a mechanism called virtual sequences. All these sequence capabilities promote reuse as described below.

An OVM sequence mechanism is comprised of three entities:

- Sequence(s)
- Sequencer: a verification component that mediates the generation and flow of data between the sequence(s) and the sequence driver. The sequencer has a collection of sequences associated with it called sequence library.
- Driver: a verification component that connects to the pin-level interface of the DUV. Drivers include one or more transaction-level interfaces that decode the transaction and drive it onto the DUV’s interface.

4.1 Sequence

Sequence is a construct that generates and drives transfers (or sequence items) to a driver via a sequencer. This is referred to as flat sequences. A sequence can also call other sequences. This is referred to as hierarchical sequences. Figure 14 and 15 showed the example of a flat sequence and a hierarchical sequence respectively.

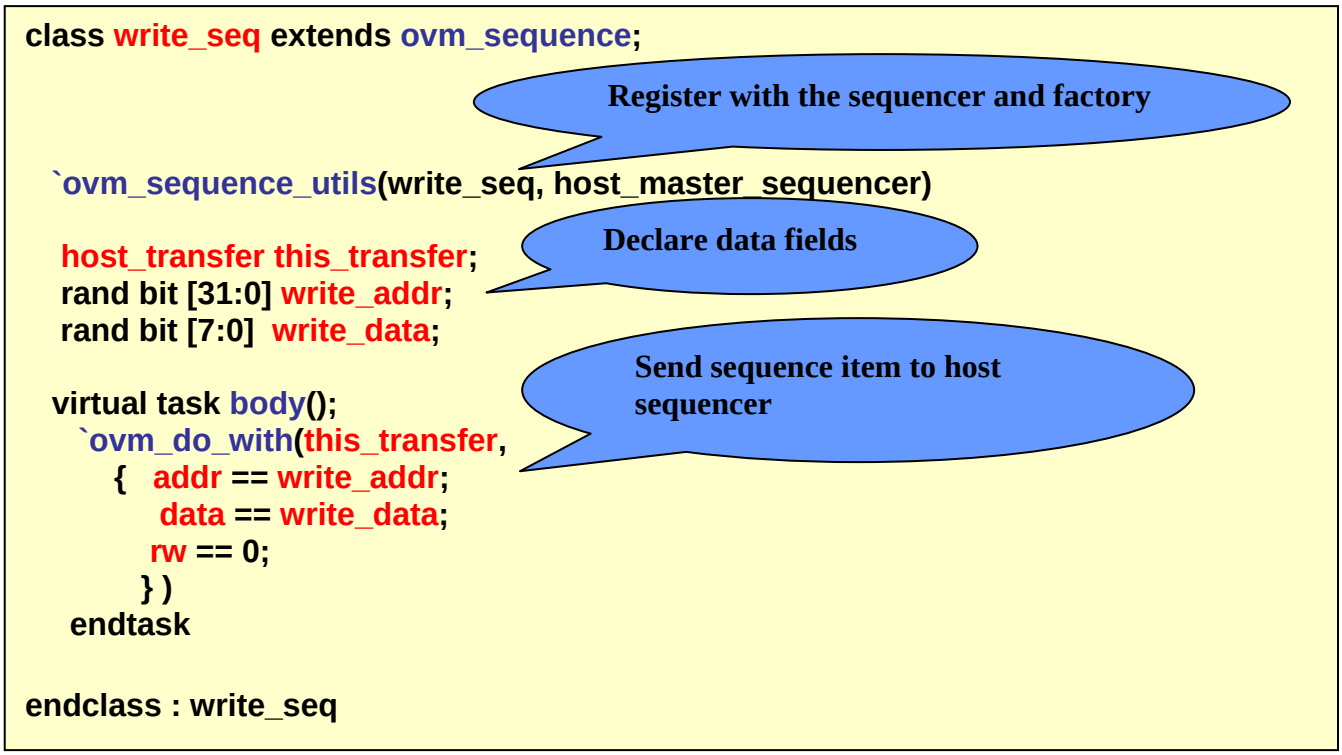

Figure 14 Example of Flat Sequence

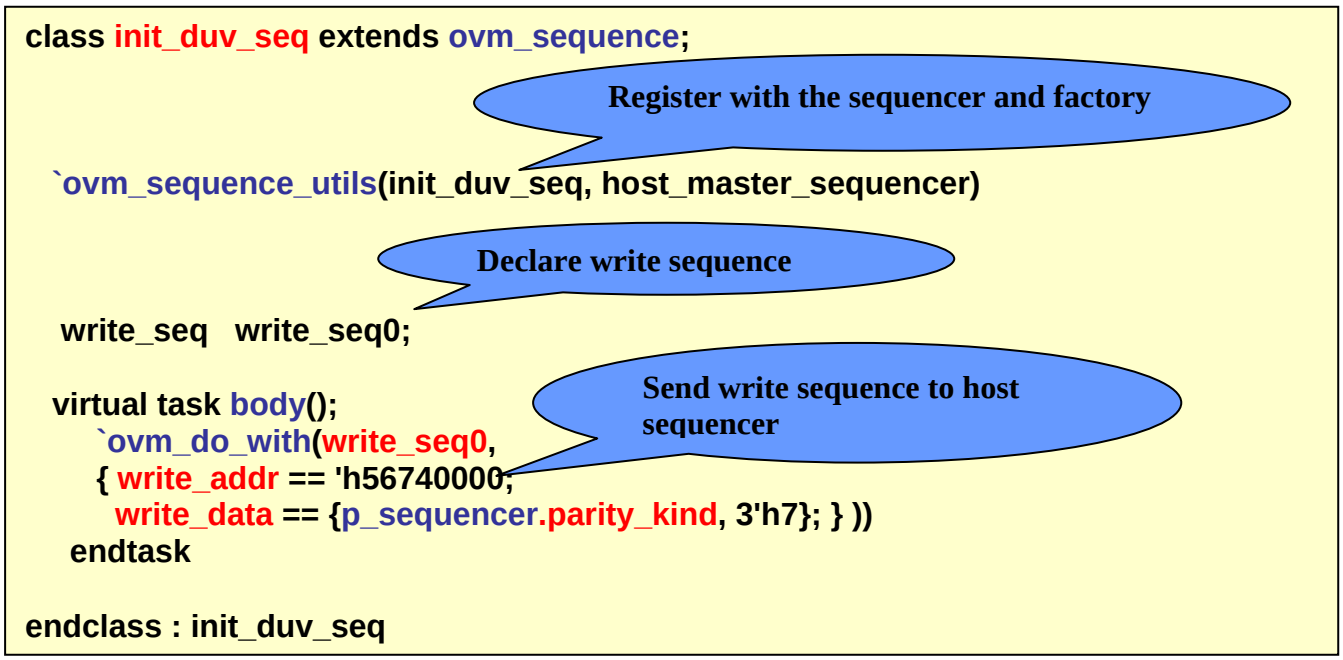

Figure 15 Example of Hierarchical Sequence

4.2 Sequencer

The PW Router DUV testbench defined two sequencers, a host sequencer and a packet sequencer. Figure 16 shows the sequence item used by packet sequencer, packet_transfer. It is derived from the ovm_sequence_item class and contains the addr, data, and parity fields. These fields are declared as random using SystemVerilog *rand* keyword. A SystemVerilog constraint block named parity_error_c is also included in the pi_transfer class to prevent the randomization of the packet transfer from generating invalid parity value. The packet_transfer class is registered into the factory using OVM utility macro to take advantage of the OVM configuration mechanism.

```
class packet_transfer extends ovm_sequence_item;

 rand bit [31:0]        addr;
 rand bit [7:0]         data[];
 rand bit          parity_error;
       …
constraint parity_error_c {parity_error == 0;}

`ovm_object_utils_begin(packet_transfer)
   `ovm_field_int (addr,  OVM_ALL_ON)
   …
   `ovm_object_utils_end

endclass : packet_transfer
```

Figure 16 Packet Transfer Class

The host sequencer includes a library of sequences that includes the write_seq and init_duv_seq. The "default sequence" is the sequence that starts when the sequencer enters the OVM run phase (or when simulation time starts). For the host sequencer, the "init_duv_seq" sequence is assigned as the default sequence. See Figure 147 below.

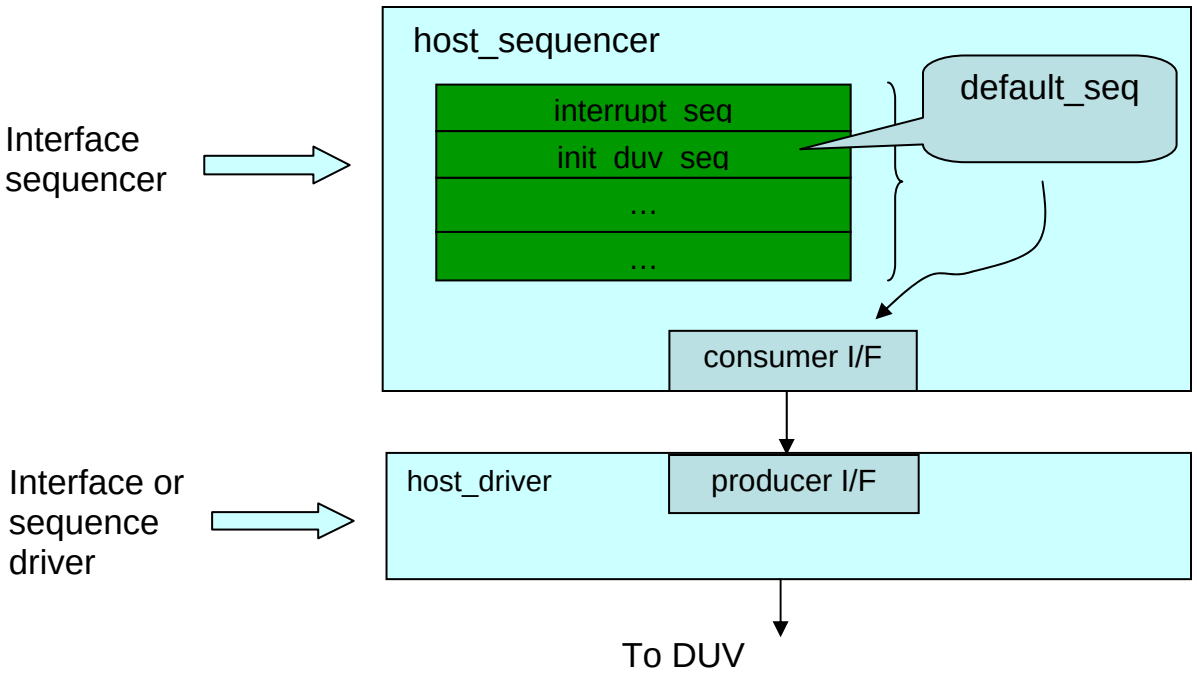

Figure 147 Sequencer/Driver Interface

The host sequencer operates in pull mode. In pull mode, the sequencer presents the transaction to the driver and the driver is responsible for pulling the sequence item out of the sequencer and drives it onto the physical busses.

4.3 Driver

As just stated, the driver is responsible for pulling transactions from the sequencer and driving them onto the pin interface of the DUV. In the example codes shown in Figure 158, the host_driver class, which is inherited from the ovm_driver class, executes a forever loop inside the OVM's run() task. The forever loop calls a blocking task get_next_item() . This task retrieves the next sequence item from the host sequencer. Next, the sequence item is cast into a host transfer and calls out a drive_transfer() task. This task decodes the host transfer and drives the data onto the pin interface of the DUV. After the data is driven out, the item_done() task is called to signal the sequencer that the transfer has finished.

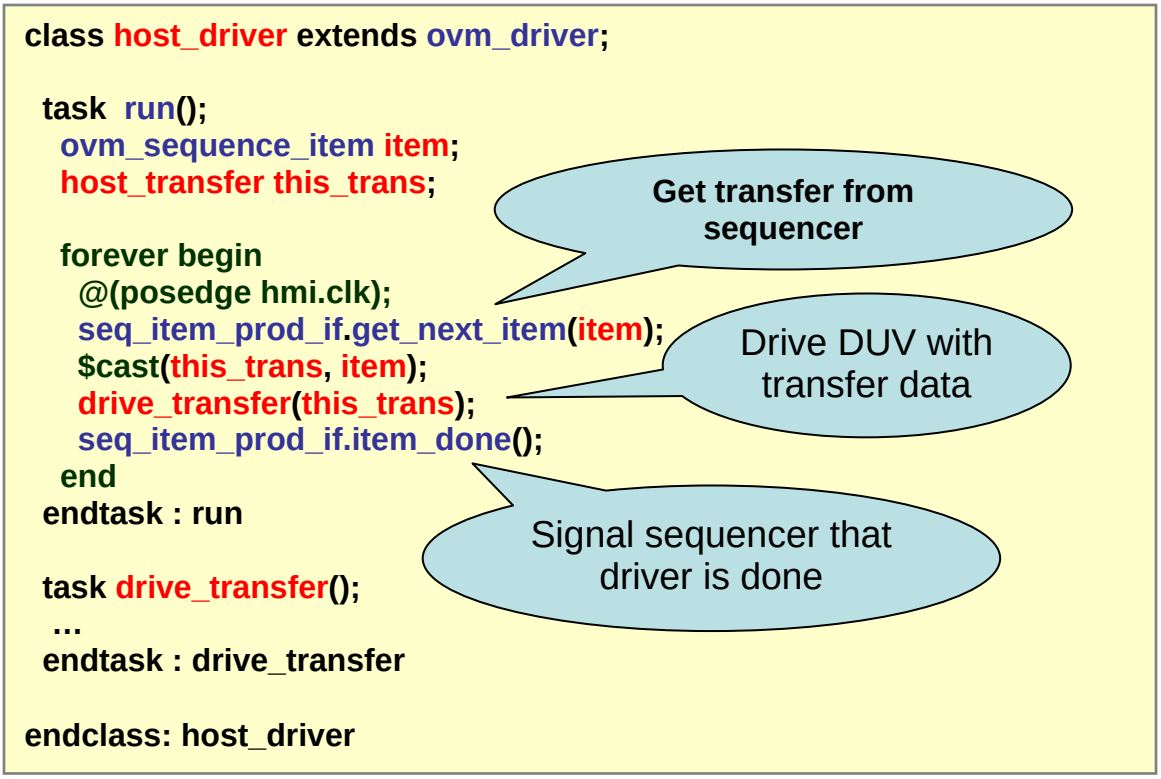

Figure 158 Driver

Notice that the sequence driver communicates through a special TLM consumer/producer interface. This allows different kinds of drivers to easily be swapped using the same sequencer.

4.4 Virtual Sequencer and Virtual Sequence

The PW Router design requires the host to initialize the DUV before routing packet traffic. Additionally, while packet traffic is flowing, the host interface needs to service interrupts. Therefore, we need to coordinate control of stimuli on both the host and packet interfaces. OVM virtual sequences provide this type of coordination.

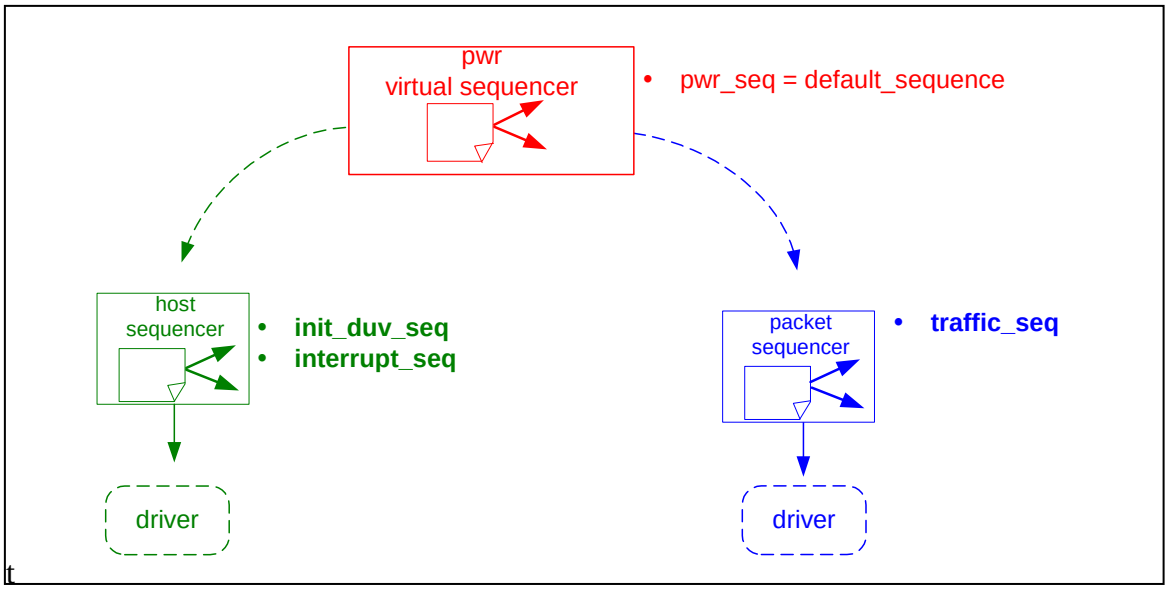

Figure 169 PWR Virtual Sequence Architecture

Figure 169 shows a graphical view of the PW Router Virtual Sequencer and its connections to the downstream host and packet sequencers. It is assumed that the virtual sequencer is the initiator component and the downstream sequencers are the target components. The virtual sequencer initiator has the ability to call out the library of sequences in the target sequencers. For example, the PW Router Virtual Sequencer may invoke the init_duv_seq and/or interrupt_seq defined in the host sequencer. Similarly, the PW Router Virtual Sequencer may also invoke the basic_traffic_seq in the packet sequencer. Note that this is a simplified list of sequences for the intent of this paper. Our sequencers have additional sequences and the built-in sequences as described in the OVM Reference Manual[iii].

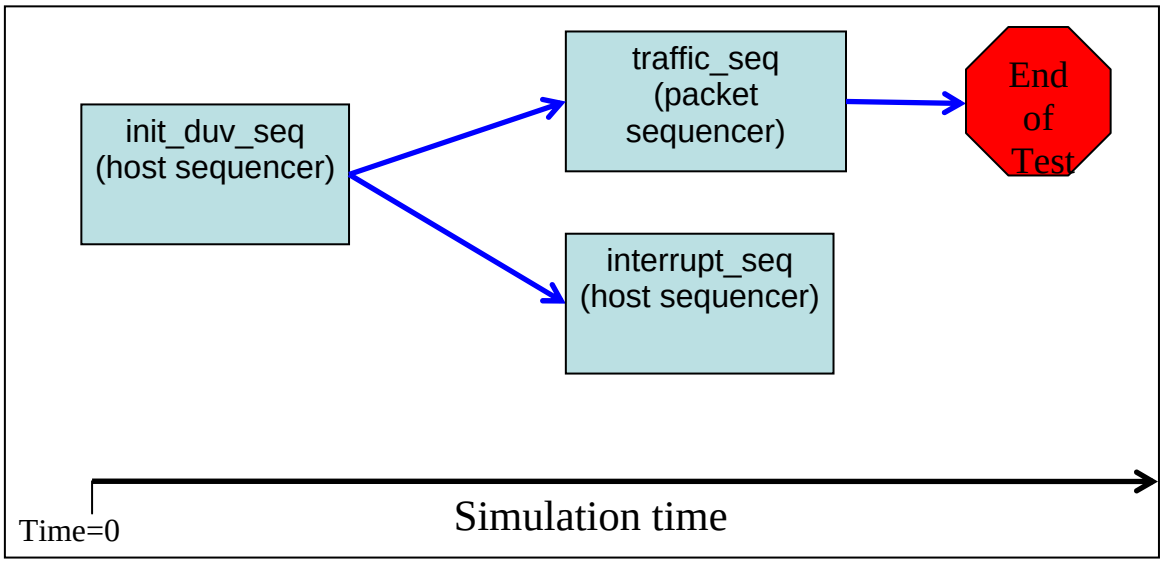

Figure 20 PWR Virtual Sequence Progression

Figure 20 depicts the sequence progression we used in PW Router OVM Testbench. The testbench boilerplate pwr virtual sequence first initializes the DUV using the host sequencer's init_duv_seq and then invokes a traffic sequence using the packet sequencer in parallel with the host sequencer's interrupt sequencer.

Figure is example code snippet of the PWR Virtual Sequence progression. Virtual sequence "pwr_seq" extends from the ovm_sequence class. It instantiates two host sequences, i.e., init_duv_seq and interrupt_seq, and one packet sequence, traffic_seq. In the body() of the virtual sequence, we first invoke the host's init_duv_seq using OVM virtual sequence macro called `ovm_do_seq. When init_duv_seq finishes, the packet and interrupt sequences are concurrently invoked using the `ovm_do_seq macro.

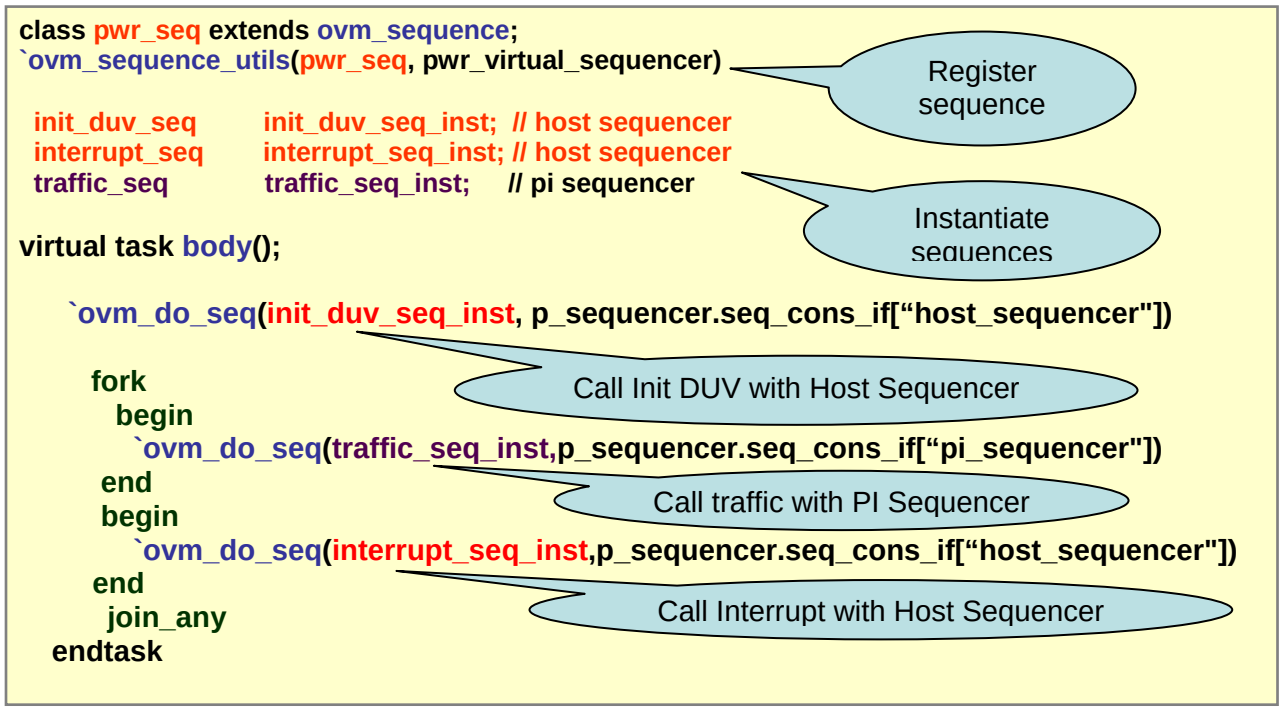

Figure 21 Virtual Sequence

5 The OVM Factory - Parity Error Example

The OVM factory is a powerful mechanism that allows test writers to override the default behavior exhibited in the testbench. The factory and configuration mechanism can both override testbench behavior but have different charters. The primary focus of configuration mechanism is to allow various layers of the testbench to overwrite default field values in a top-down manner during the build phase. The factory gives users the ability to override OVM objects during both the build and run phases.

An OVM factory is a static singleton object. When OVM objects are created in the testbench, they may be registered into the factory. Test writers can derive their own OVM objects and then perform type overrides (globally or on a particular instance) of those OVM objects in the testbench. This methodology is completely non-intrusive with regard to the testbench code. The test writers may change the behavior

of an OVM object by overwriting virtual functions, adding properties, as well as defining and adding additional constraints.

The following example shows how a test writer can intelligently override the PW Router Testbench's default behavior using the factory. Packet transactions, by default, are constrained to send only legal parity packets (see Packet Transfer Class in Figure 16). However, the test writer can use inheritance and the factory to force illegal parity data. Figure 22 shows a new class called my_error_traffic_seq that inherits from the parent class traffic_seq. The `ovm_sequence_util macro registers my_error_traffic_seq with the factory and then adds it into the pi_sequencer's library.

Inside the body() of the my_error_traffic_seq, the `ovm_create macro is used to obtain a reference to the pi transfer. Next, the parity_error_c constraint which forces the pi transfer to generate only good parity is shut off. Then, the `ovm_rand_send_with macro is called to send the sequence to the sequencer/driver with the pi transfer constrained to always have parity error.

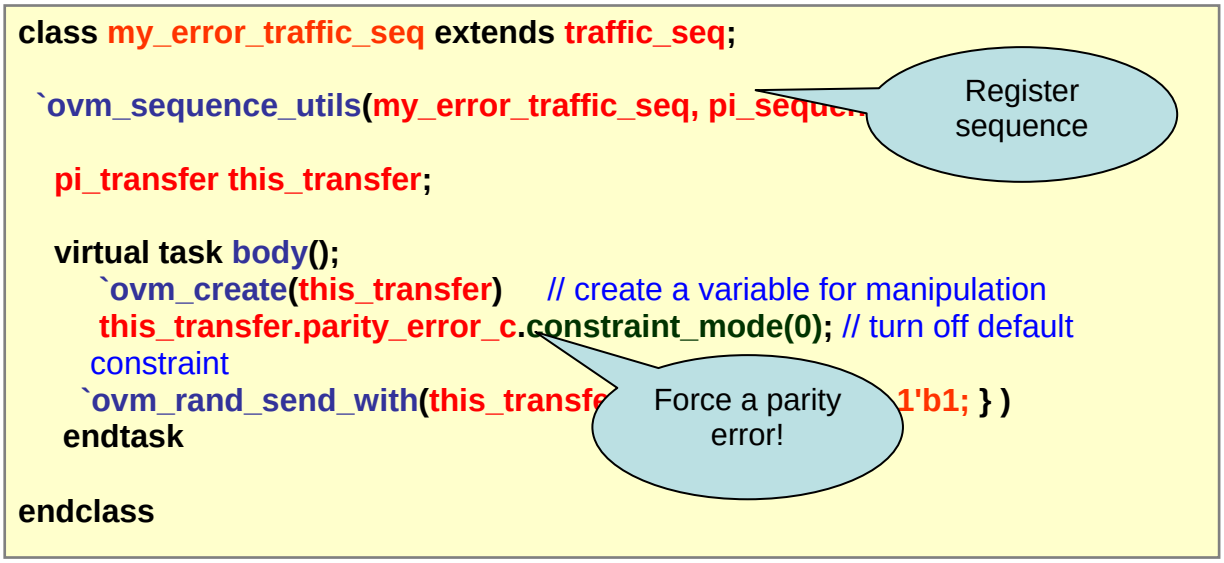

Figure 22 Packet Error Transfer

Figure 23 shows how the test writer may use the OVM factory methods set_type_override() or set_inst_override() to force the testbench to send the error traffic. set_type_override() replaces the type traffic_seq with my_error_traffic_seq globally while set_inst_override() method overrides a type based on some component's instance in the testbench hierarchy.

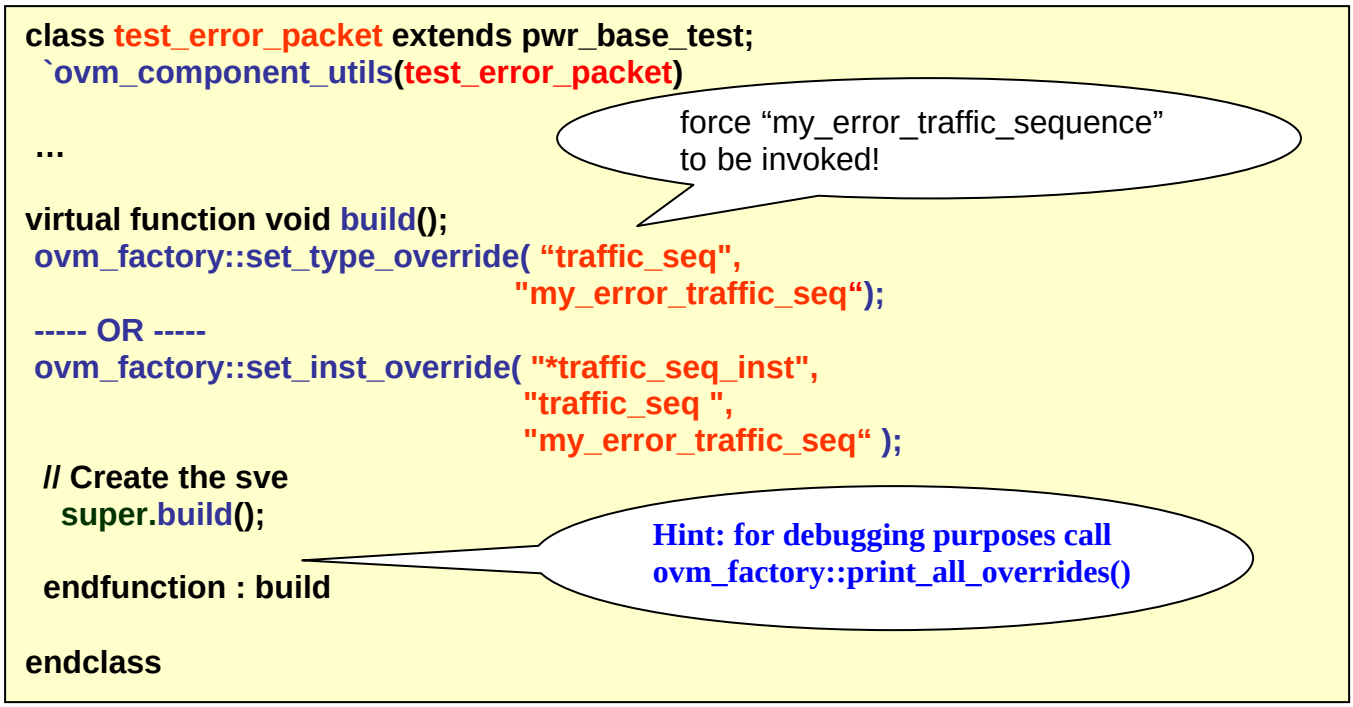

Figure 23 Parity Error Factory Overrides

6 Automating OVM Testbench Generation

As we have shown in previous sections, well-structured OVM verification components are highly reusable. However, OVM usage is quite open ended. Teams may implement their code using an approach that may be more geared towards either the AVM or URM style. We found that just implementing a "best-practice" OVM testbench framework is a time consuming task.

Furthermore, the OVM methodology lacks recommendations for directory structure, file-naming conventions and coding styles. Typically, these types of guidelines are beyond the scope of standard verification methodologies such as OVM. Organizations usually have their own strategies in place to handle them.We also found organizations have difficulty deploying their "best-practice" usage uniformly throughout the entire verification organization which significantly affect the reusability of their testbenches.

To overcome these deployment obstacles we developed a Template Generator (TG) tool that automatically generates a testbench based on templates. Figure 24 shows the flow of the TG Tool. We created a complete set of generic OVM templates that feed into the TG. These templates were implemented using our "best-practice" techniques for monitors, sequencers, sequence libraries, drivers, agents, virtual sequences, interface/module OVCs, and SVEs. The template generator builds up an entire OVM testbench that includes a makefile and a dummy test that allows teams to compile all the codes out of the box using Cadence or Mentor simulators. The TG allows teams to control the name and number of OVCs they want to generate.
Moreover, organizations may easily customize the templates for changes such as coding styles, naming conventions and so on. TG helps teams across the entire organization to deliver testbench code that has the same "look-and-feel". This significantly speeds up the testbench development and improve the

productivity. In addition, TG is also capable of merging changes into previously generated code in the case when the teams decide to modify their “best practice” approaches.

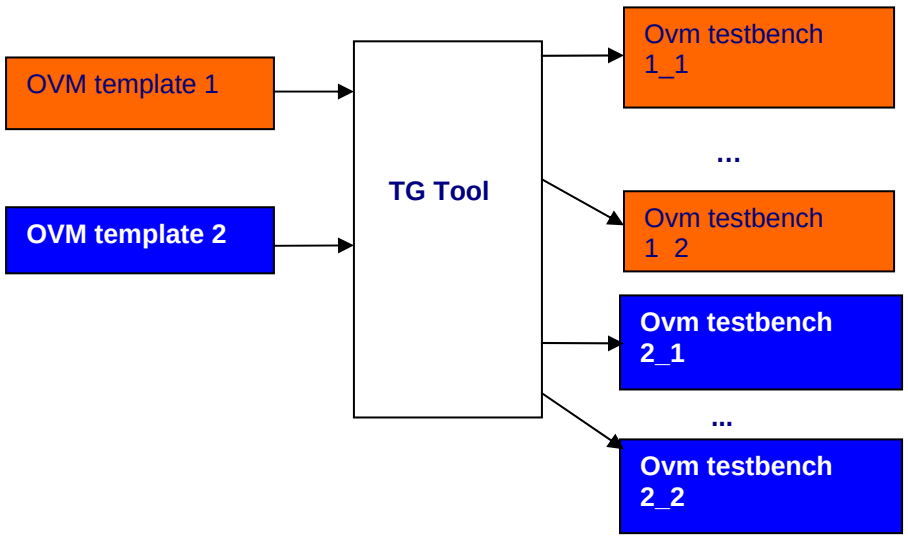

Figure 174 Template Generator

7 Conclusion

The migration effort from AVM and URM to OVM is a relatively easy process. We were able to rerun our AVM 2.0 code using the OVM libraries with no changes. We ran into several minor syntax issues during our URM to OVM code conversion exercise.

More important than the migration effort is for the verification architects to understand which OVM features their team needs to utilize and designing a testbench architecture that is sufficient to hit the coverage goals. OVM has rich features that greatly help with reuse such as the configuration mechanism, factories, TLM and sequences. However, employing these advanced features come at a cost of extra effort. These advanced features may seem to be an overkill for what is currently needed to hit a project’s coverage goals but it is our experience that the small upfront cost pays dividends down the road.

Implementing OVM code is often difficult due to the complexity of debugging macros. However, the open source gives the users the ability to debug issues much deeper with a clearer understanding. It is vital that users do not stray away from the OVM structure shown in the xbus example in the OVM library and the OVM User Manual.We ran into numerous difficult debug issues while experimenting with changing the order of the configuration override calls, component creation and build steps. Overall, we found the macros help make the code more intuitive and readable. Additionally, the configuration wildcard “field matching” capabilities greatly reduced the configuration code compared to previous eRM (e Reuse Methodology) coding efforts.

---

[i] AVM Cookbook Copyright © 2007-2008 Mentor Graphics Corporation
[ii] Step-by-Step Functional Verification with SystemVerilog and OVM
[iii] OVM Reference Manual for OVM 1.1