Module- or Class-Based URM? A Pragmatic Guide to Creating Verification Environments in SystemVerilog

Paradigm Works, Inc.

Dr. Ambar Sarkar

Session # 2.15

Presented at

cadence designer network

Silicon Valley 2007

Module- or Class-Based URM? A Pragmatic Guide to Creating Verification Environments in SystemVerilog

Dr. Ambar Sarkar

Paradigm Works, Inc.
ambar.sarkar@paradigm-works.com

ABSTRACT

Given the increasing adoption and maturity of SystemVerilog as a viable HVL (High-level Verification Language), Cadence has released two flavors of the Incisive Plan-to-Closure methodology: 1) SystemVerilog Module-Based Universal Reuse Methodology, and, 2) SystemVerilog Class-Based Reuse Methodology. Both flavors draw heavily from the proven e Reuse Methodology (eRM). According to the documentation, the module-based approach is targeted towards design teams who quickly want to get started with the creation of block- and chip-level environments. On the other hand, the class-based approach is geared towards verification teams who can take advantage of advanced object-oriented software techniques to implement constrained-random, plan-driven verification environments.

From a user perspective, this begs the question of whether to use a class-based approach or a module-based approach. Both design and verification teams are intimately involved in a verification project. So, should one recommend creating two separate environments, one for the design team to do module level testing, and leave the heavy-duty verification effort to the verification team using a class-based approach? Alternatively, does it make sense to do it all in one approach? Even better, should the design team create a module-based environment, which is then reused by the class-based approach? To answer such questions and to gain a deeper understanding of how each approach works, we applied both to develop verification environments for a realistic design example. The example consists of a packet router implementation that exhibits features of a typical verification project. In this paper, we will report our experiences and share the lessons learned.

Table of Contents

Table of Figures

1. Introduction

Schedule pressures in current functional verification projects often dictate that both design and verification teams contribute actively to the effort instead of just the latter. Typically, the design team members are required to do some early verification of their own blocks, followed by some verification on blocks developed by others, as well as creating some corner case scenarios down the road. Unfortunately, for the most part, these designer created test suites are done in stand-alone and ad-hoc environments, not easily integrated with the testbench put together by the verification team. This lack of integration is often due the difference in languages, tools and methodology adopted by each team as well as the learning curve involved in understanding each other's approach.

There are two key developments taking place in the industry today that should help address this lack of integration between the verification codes developed by the both teams. First is the increasing adoption of SystemVerilog [1] as a common language to be used by both design and verification. While each team may prefer using some specific SystemVerilog language features to the other, adopting this common language makes it much easier to interpret, create, and execute with the code developed by the other team. However, just having a common language is just the first step.

The second key development is the availability of the standardized verification methodologies. In addition to a common language, a common coding approach is needed so that the components developed can work efficiently together and be reused across projects. Such a verification methodology is offered today by Cadence in the form of Incisive Plan-to-Closure Methodology (IPCM) [2][3].

IPCM comes in two flavors: 1) SystemVerilog Module-Based Universal Reuse Methodology (MB-URM) [2], and, 2) SystemVerilog Class-Based Reuse Methodology (CB-URM) [3]. Both flavors draw heavily from the proven e Reuse Methodology (eRM). According to the documentation, MB-URM approach is targeted towards design teams who quickly want to get started with the creation of block- and chip-level environments. On the other hand, the CB-URM approach is geared towards verification teams who can take advantage of advanced object-oriented software techniques to implement constrained-random, plan-driven verification environments.

Given these two developments, we wanted to explore how both the design and verification teams can create a synergistic environment for verification. In this paper, we present a case study where we apply both the MB-URM and the CB-URM approaches in creating a verification environment using a realistic design. Specifically, we created a hybrid environment, which uses both URM approaches and shows how components developed in either methodology can be executed together while reusing elements from their counterparts.

2. Choosing the methodology

Given the two approaches for IPCM, we had three possible candidates for verification methodology adoption. The first was to adopt the MB-URM for both designers and verification teams. This made sense from ease of use for the design team, but the verification team would miss many useful and sophisticated features available in the CB-URM. For example, while the MB-URM supports creating sequences, it does not offer the rich set of operations supported in the CB-URM. The second candidate was to adopt CB-URM for both teams. This approach, while very powerful as a verification methodology, would require significant investment in time and effort for the design team, and probably require a different skill-set.

So a third candidate, a hybrid approach was chosen. In this, methodology, the design team would create its environment and tests using the MB-URM. At the same time, the verification team would develop its environments and tests using CB-URM. Next, these two types of environments and the components comprising them would be brought together to interoperate under a hybrid environment.

To determine whether such an approach would be successful in a real project, it was expected that the integrated approach would support the following activities:

a. The design team should be able to easily create module based environments and stand-alone tests as needed.

b. Both the design and the verification teams should be able to execute their verification components together. For example, a MB-URM monitor for the host interface should be able to run with a CB-URM monitor for the packet interface.
c. It should be able to run test sequences developed using MB-URM in parallel with those developed in CB-URM. For example, one should be able to create system-level test scenarios where the initialization of the chip is done using a sequence implemented in MB-URM while a traffic pattern is generated using agents developed in CB-URM. In other words, one should be able to create super sequences that had sub-sequences from both MB-URM and CB-URM.
d. Reuse all self-checks implemented in either environment.

There were several challenges in implementing the hybrid methodology. First, there are two further flavors of the MB-URM: the simple testbench environment and the general testbench environment. It was not clear whether to use the former approach that does not have notion of agents or sequences, or to use the more sophisticated latter approach, which allows one to create self-contained environment modules that support libraries of sequences. The former is straightforward, but seems ad-hoc compared to the more structured approach of the latter. On the other hand, the latter approach requires a little bit of extra work in defining the sequence driver and creating the additional agent and environment modules.

Another obvious challenge is to make the objects represented as SystemVerilog classes access and interact with objects that are module based. For example, a BFM is represented as a SystemVerilog module, and it communicates with the DUT and the other verification elements using SystemVerilog interfaces. The analogous element in the CB-URM is an instance of a user-defined SystemVerilog class extended from the URM base class *urm_bfm*. This CB-URM BFM object uses the SystemVerilog *interface* construct to connect to the DUT, but uses TLM[4] ports to connect to other verification objects. Another challenge was to determine exactly how the sequences developed in either environment can be made to interact; the MB-URM uses task calls, while the CB-URM uses macros, which in turn use class method calls.

There were some tool related challenges to overcome as well, since both MB- and CB-URM use different base libraries, so it was not clear if they would even compile together. However, this is very likely to be a non-issue, as we were using non-production versions of the tools, and we assume that Cadence will ensure interoperability of the base libraries going forward.

The following case study describes how we addressed the challenges.

3. Case Study

In this section, we first describe the design under verification. We then describe how the MB-URM components were developed for one of the interfaces; next, the development of the complementary components using CB-URM is presented. Finally, we describe how the hybrid environment was constructed.

3.1 Design under verification: the PW Router

To demonstrate the hybrid approach, we chose an in-house router implementation. The router, called the PW Router (PWR) is a one-input, three-output packet forwarding engine that routes packets based on the address and priority fields in the packet. The PWR has an interface to the external memory that it uses to store packets and host CPU uses for scratch data. The host interface also allows the CPU to access various internal registers that control the device and provide feedback on its status. Figure 1 shows a general block diagram of the device.

For the purpose of this exercise, we assigned the verification of the host interface to the MB-URM approach. Next, we assigned the verification of the packet interface, the main datapath, to the heavy-duty CB-URM approach. In a real situation, this assignment would make sense as verifying the host interface is usually the simpler task, while verifying the main datapath and the related control logic would be the task of the verification team.

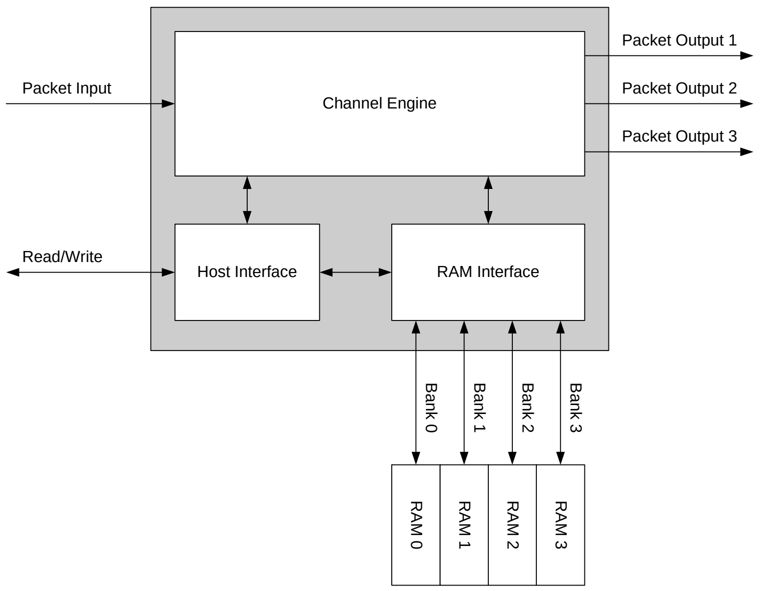

Figure 1. Paradigm Works Router (PWR)

3.2 MB-URM environment for the host interface

The MB-URM verification environment for the host interface was created by following the guidelines in **Error! Reference source not found.**. As recommended, a SystemVerilog wrapper was created around the DUT to access the Verilog ports as SystemVerilog interfaces. Second, a transaction type was defined for the host interface transactions using the SystemVerilog *packed struct* to facilitate cross-language communication. However, to take advantage of the randomization feature of SystemVerilog, a class type is declared that defines the packed struct object as a randomizable member. This allows one to create randomized transactions. In general, having a module-based approach does not mean one should not use classes at all, it is just that the key components are defined as modules. The follow shows how the packed struct pwr_hi_request_s is defined first and then the transaction class pwr_hi_transfer_data_item_c is defined for generating randomized transactions:

```
typedef struct packed
{
        bit [31:0] address;
        bit [7:0] write_data;
        bit [7:0] read_data;
        mem_transaction_type_e request_type;
} pwr_hi_request_s;


class pwr_hi_transfer_data_item_c {
  rand pwr_hi_request_s req;
  constraint c_address { req.address inside {[32'h56740000:32'h5674001B]};}
  ...
}
```

Next, the BFM is defined, which converts transactions to DUT signal interaction and vice versa. The BFM uses SystemVerilog interface to communicate to the DUT as well as the rest of the verification objects. In addition, monitor modules are defined that observe the transactions on the host interface and check for both protocol and data violations.

After having defined the transaction class, the BFM, and the monitors, one can create simple tests in MB-URM. However, tests scenarios created in this manner are not easily reusable and are harder to maintain as they grow in number. Based on the recommendation in **Error! Reference source not found.**, to promote reuse, an agent module was therefore created as shown in **Error! Reference source not found.**. As part of this agent, a sequence driver was defined (pwr_hi sequence driver). The sequence driver works by invoking a sequence task which in return creates a pwr_hi transaction that is driven by the pwr_hi BFM to the DUT.

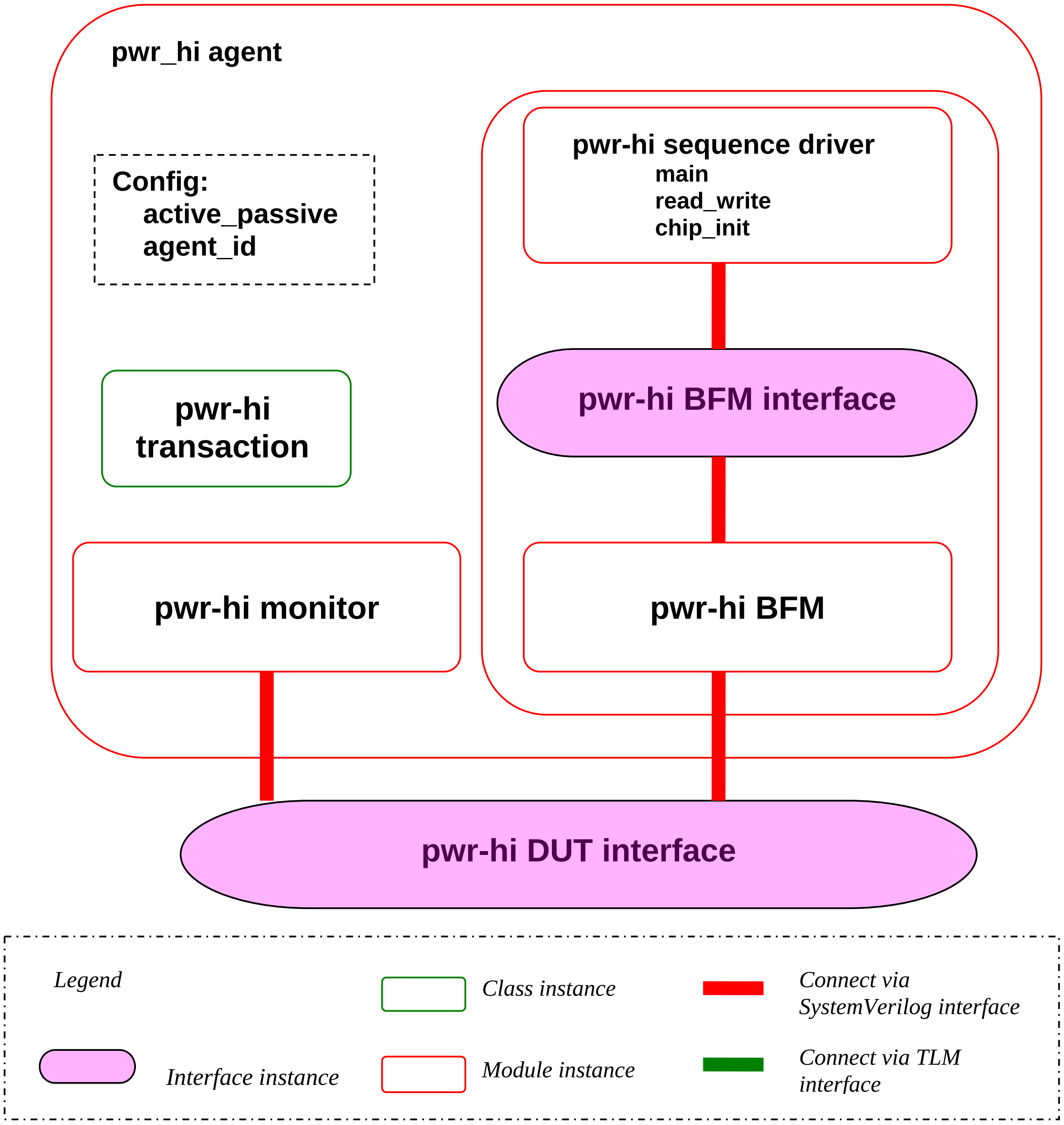

Figure 2 Module based agent for host interface

The sequence driver also defines a sequence library, which is essentially a collection of tasks, each representing an individual sequence. For example, three sequences are defined, the main sequence which sends in a random pwr_hi transaction, the read_write sequence, which sends in a random write request followed by a read to the same address, and a chip_init sequence that initializes the chip. While creating the stimuli as sequences takes a bit of an effort, it helps create a modular verification environment such as the one shown in Figure 3.

3.2.1 Stand-alone env module for the host interface

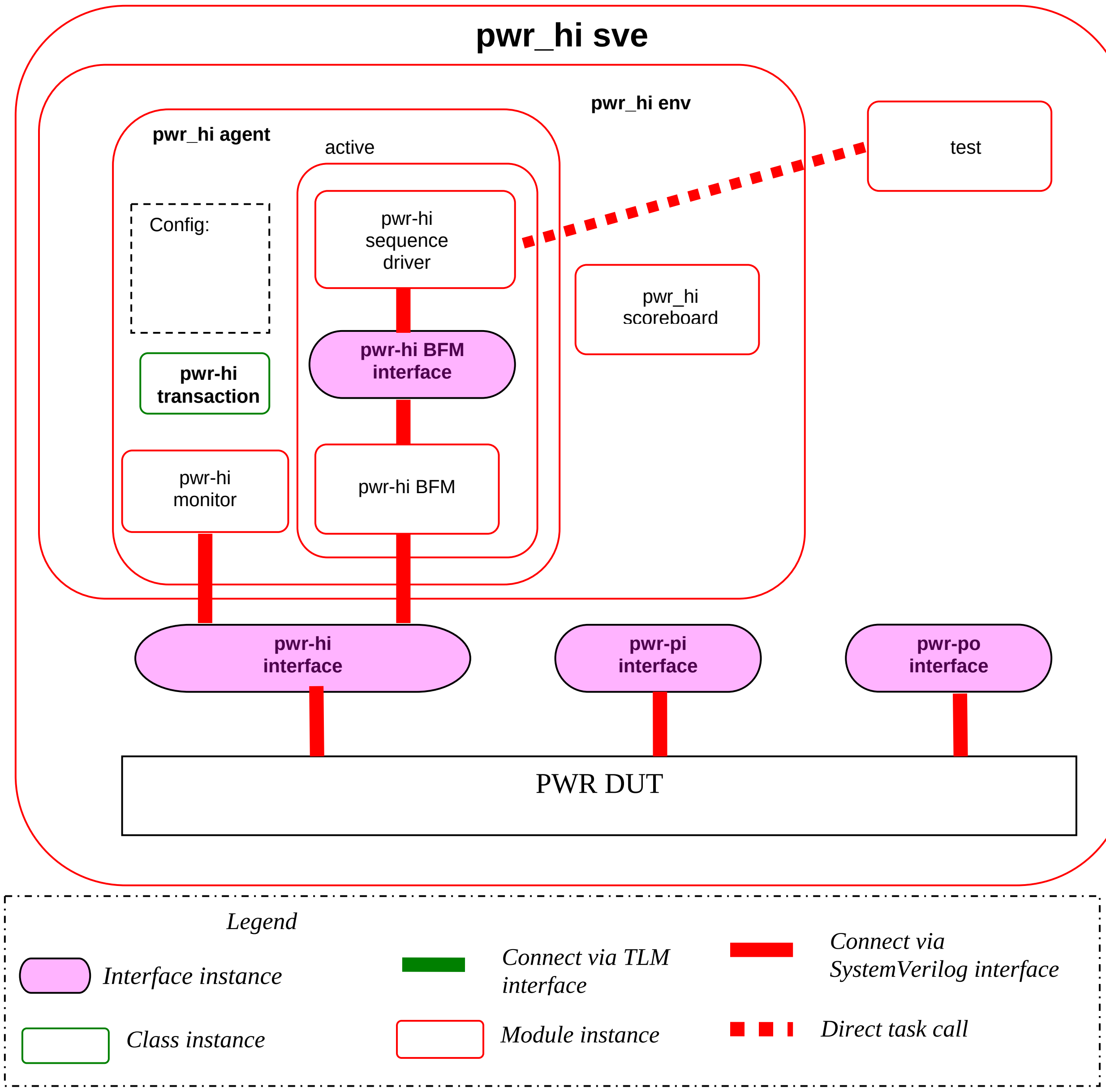

Figure 3 Stand-alone module based verification environment for the host interface

Figure 3 shows how the whole host interface environment module is structured, and how its tests are invoked. Typically, a test calls a specific sequence task, and the checking of the DUT behavior is implemented using scoreboards. The protocol checking can be done in the pwr_hi monitor, while any data checking can be done either in the test or by creating separate scoreboards as recommended in MB-URM. The key observation here is that the pwr_hi environment module is the reusable unit, and any sequence and checking implemented with it is reusable in a wider context. In other words, any stimulus generation to be reused should be captured as a sequence. Any self-checking to be reused should be captured either as a protocol check or as a data check implemented with the env.

3.3 CB-URM environment for the packet interface

Figure 4 shows a high-level diagram of one of the packet interface verification components, based on the CB-URM. While there is an apparent similarity in how the agent is structured, there are some clear differences between the MB- and CB-URM approaches. In addition to these structures being defined as classes in CB-URM as opposed to modules in MB-URM, the interface between the sequence driver and the BFM is implemented using TLM$^{\text{Error! Reference source not found.}}$, since now the communication is between two SystemVerilog classes and not modules. Another difference is in how the sequences are defined; in the MB-URM, they are defined as SystemVerilog tasks, whereas in CB-URM, they are defined as classes derived from the urm_sequence base class.

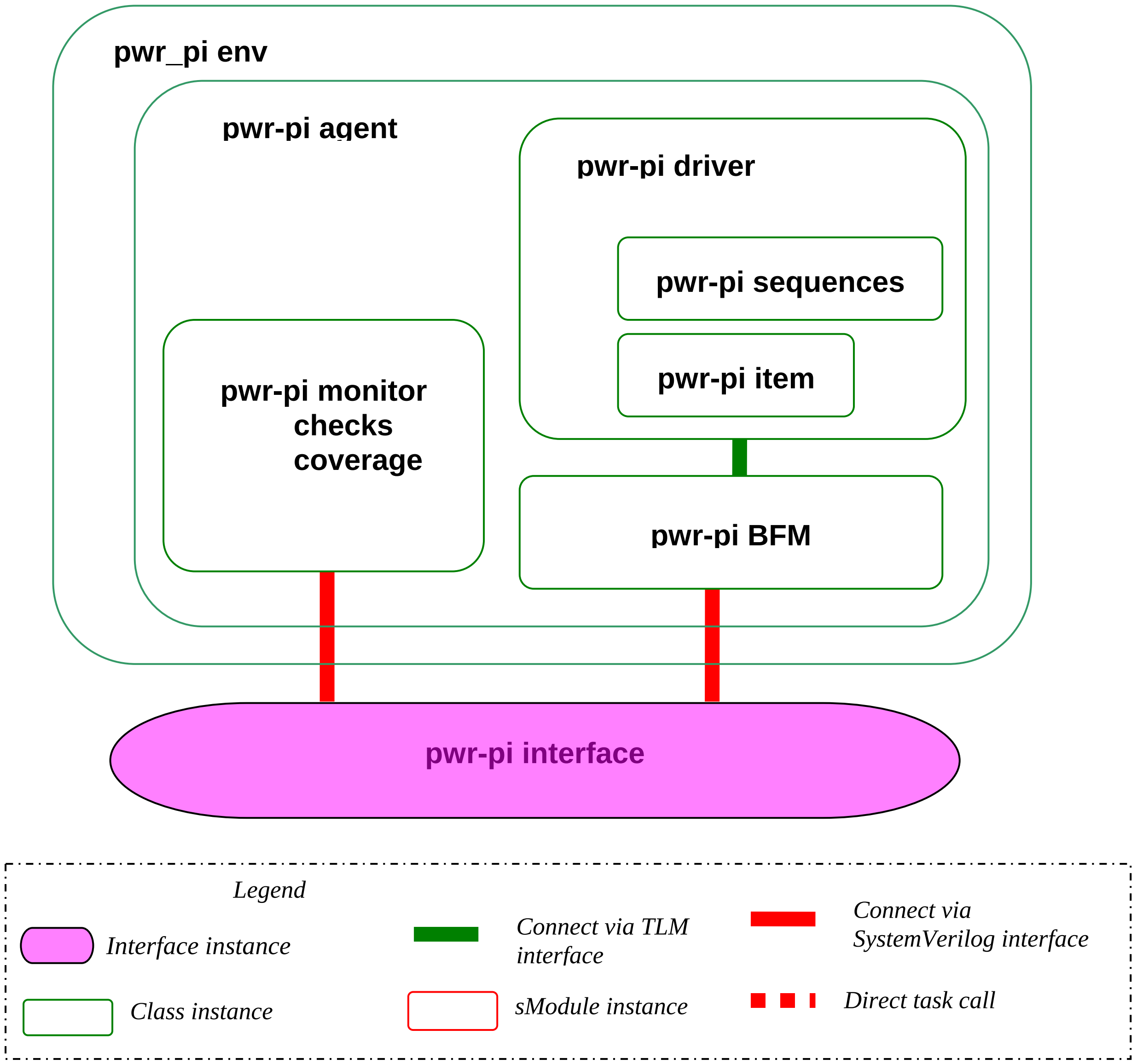

Figure 4 Class based environment unit for the packet input interface

3.4 Hybrid environment for the router

Figure 5 shows how the host interface, which is developed using MB-URM, is put together with the rest of the CB-URM based verification environment to create a hybrid environment.

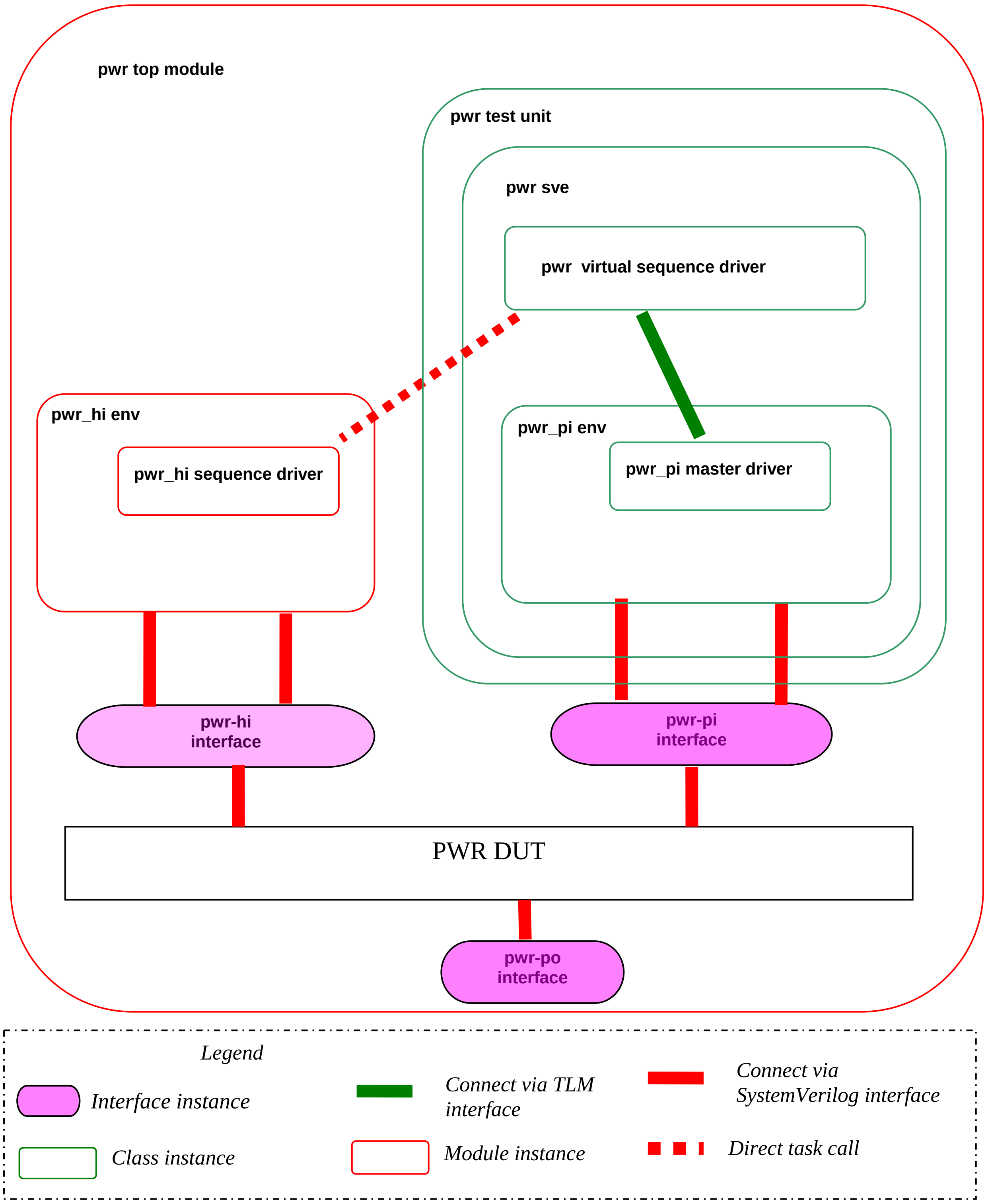

Figure 5 Hybrid class and module based environment for PWR

The following describe the steps needed for the integration of the MB-URM and CB-URM environments.

Step 1. Instantiate the MB-URM environment modules as part of the top-level CB-URM module.
This is just a simple instantiation of the MB-URM environment as shown in the following code snippet.

```
module pwr_tb_top;
....
// The verification environment module
            pwr_hi_env_m #( .MASTER_INSTANCES(1),      // 1 ACTIVE masters
                            .masters_active_passive({ACTIVE, ACTIVE}),
                            .DEFAULT_COUNT(0))      // sequence drivers off by default
     pwr_hi_env(pwr_hi_if);
...
endmodule //
```

Step 2. Synchronize the start of the tests
In the MB-URM environment, the top-level module makes a call to the *start_test()*, which is defined in the Cadence's urm_utils_pkg package. Since the top-level module of the MB-URM is not included in the hybrid environment, we need to make that call explicitly. This can be done in parallel with the *run_test()* call made in tine CB-URM top-level module to start the CB-URM environment.

```
module pwr_tb_top;
...
 initial
            begin
                        fork
                                    begin
                                                #1;
                                                $display("Generating UB-URM start_test");
                                                urm_util_pkg::start_test();
                                    end
                                    run_test();
                         join
             end
end
```

Step 3. Create combined sequences as needed.

The following code snippet shows how to create hybrid sequences using both CB and MB sub-sequences. The basic idea is to access the sequence generation tasks in the MB environment by using the $root as a way to traverse the hierarchy.

```
class hybrid_seq extends urm_sequence;
            ...
            `urm_sequence_utils(hybrid_seq, pwr_pi_master_driver)
             send_pkts_seq pkts2port0; // Sequence defined in CB-URM
            virtual task body();
            begin
                        ...
                        $root.pwr_tb_top.pwr_hi_env.masters.agent[0].active.seq_driver.init();
                         `urm_do(pkts2port0)
      end
```

```
    endtask : body
endclass : hybrid_seq
```

In addition, there were a few steps needed to compile both environments together. We also had to comment out one of the macro calls *`urm_set_thread_seed* related to the random number generation the in the MB-URM environment for the host interface. We also did not define any SystemVerilog package structures within the CB-URM code as we ran into compile issues. However, we believe these steps and workarounds will be unnecessary in the production version of the tool.

4. Conclusion

In this paper, we propose a hybrid methodology that combines both the MB-URM and the CB-URM approaches. We demonstrate how such methodology allows design teams to contribute substantially in creating and maintaining verification environments. This is achieved by the reuse the stimulus generation and the response checking developed by design team members by the verification team. Based on this result, we recommend an approach where the design team uses the MB-URM to get started creating its own unit-level tests. The verification team should use the CB-URM to implement the majority of the verification effort, but reuse the MB-URM components developed as needed. Once the RTL development stage is over, the designers can contribute further by adding more verification code, as well as add more complex sequences that include those developed using MB-URM and CB-URM.

5. References

[1] SystemVerilog language reference IEEE P1800, "Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language," IEEE, 2005

[2] Incisive® Plan-to-Closure Methodology - SystemVerilog Module-Based Universal Reuse Methodology (URM), Version 6.0.1

[3] Incisive® Plan-to-Closure Methodology - Universal Reuse Methodology (URM) SystemVerilog Class-Based Implementation User Guide (Beta) Version 6.11 EA3.2

[4] TLM Transaction Level Modeling Library, Release 1.0, www.systemc.org