A SystemVerilog Framework for Easy Method Advice in Object-oriented Test Benches

Eric Ohana
Media Processor Division ARM Ltd

ABSTRACT

It is a common necessity, while utilizing an object-oriented verification environment - a test bench, for exercising the features of a Design Under Test (DUT), to have to modify the methods of the various classes, the verification environment comprises.

These classes can be data classes used for generating different stimuli to the DUT or architecture classes used to build the infrastructure of the test bench. Modifying the methods of those classes is needed for the implementation of the test cases, configuring the verification or debugging purposes.

The paper focuses on test benches written in the increasingly popular SystemVerilog language which has object-oriented features. The standard object orientation way, using inheritance and polymorphism is generally used for the purpose of modifying classes' methods. The paper proposes a framework where the aspect orientation concept of advice on methods, defined by [1] as a piece of code to be executed at specific points (called join points) in the method, is implemented to some useful extent. The method advice features implemented by the framework are similar in concept to these in the ***e*** Verification Language, see [2] for more details on the latter. An advantage of the framework over ***e***, is the run-time method advice feature which is also demonstrated in the paper. Aspect-oriented advice on methods, as defined above, is not currently supported by the SystemVerilog LRM.

The DUT used to develop the framework is written in Verilog RTL but this fact is by no mean a limitation on the usability of the framework.

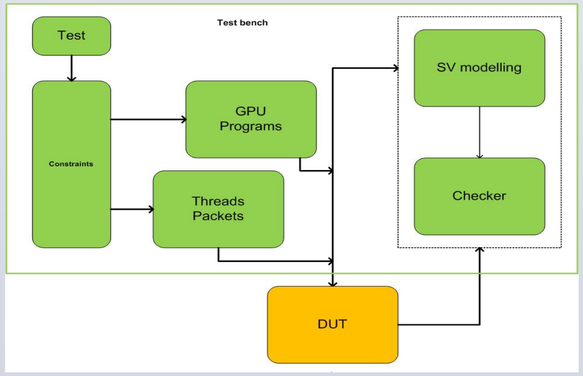
THE VERIFICATION ENVIRONMENT

An overview of the verification environment where the framework was used is shown in the diagram below.

The DUT is the execution core of a GPU program called also a shader.
The work was done as a part of an ARM MPD division project.

The diagram depicts a simplified version of the test bench. The DUT receives two main types of stimuli: 1- Randomized GPU programs 2- Randomized threads executing the randomized GPU programs.

The test case defines the randomization constraints on the types of programs and threads the DUT executes. Some of the behavior of the DUT is modeled in SystemVerilog. The checker ensures that the DUT and the SystemVerilog modeling are always aligned.



Following is a non-exhaustive list of modifications on methods for data and architecture classes needed for creating new test cases or facilitating the debugging process:

- Change the randomization results on data classes' properties instantiated in the GPU Programs block in the diagram above.
- Introduce latency cycles on a control signal in an architecture driver class belonging to the Threads Packets block in the diagram above.
- Displaying some debugging information when invoking methods in the classes of the Checker block in Figure in the diagram above.

THE FRAMEWORK

The method advice framework consists of two parts: the library part and the verification environment part:

1- THE LIBRARY PART

This part mainly defines the SystemVerilog properties a verification environment uses to activate the framework:

- typedef enum {IS_ONLY,IS_FIRST,IS_ALSO} joinPointEnum;

*joinPointEnum is an enum, which uses the same semantic and definition as in the **e** Verification Language for the method join points:*
IS_ONLY: Only the method modification is executed and not the original method
IS_FIRST: The original method is executed after the method modification
IS_ALSO: The original method is executed before the method modification
Note that if IS_FIRST and/or IS_ALSO is defined on a method along with IS_ONLY, the latter takes precedence.

- typedef joinPointEnum joinPointQueue[$];

This property is used in the class below.

- class AOPAdvice #(type T=int); … endclass: AOPAdvice

AOPAdvice is parameterized with a class type in the verification environment which uses advices on its methods.
AOPAdvice has the following static property: static joinPointQueue dataBase [string];
where the string is the name of a method in AOPAdvice parametrized class type.
AOPAdvice has methods to set (setAdvice), get (getAdvice), print (print) and reset (reset) Its dataBase property.

Finally four macros are defined:

- **`FUNCPOST(string)**
- **`TASKPRE(string)**
- **`TASKPOST(string)**
- **`TASKPRE(string)**

The usage of these macros is clarified with the verification environment part of the framework.

2- *THE VERIFICATION ENVIRONMENT PART*

The verification environment part of the framework has an impact only on the classes which want to take advantage of the method advice framework.

The other classes remain unchanged and coexist peacefully in the same verification environment.

For a given class, this part consists of:

- Defining an external function and/or an external task. The role of this external function/task is to invoke the advices on the class's functions/tasks, respectively, if some join points are defined for these class's functions/tasks. The external function is named addAdviceFunction and the external task is named addAdviceTask. Please not that this definition occurs automatically by including the framework's AOPAdviceUtils.svh file in the class definition. The actual advices for the methods, as needed by the verification engineer, are called from this external function/task.
- Adding the macros:

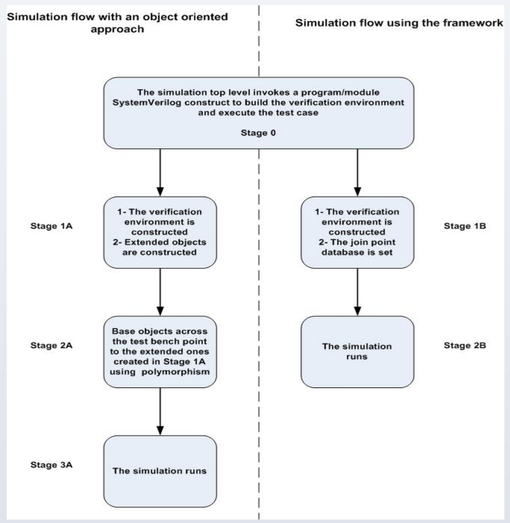
`FUNCPRE(string),
`FUNCPOST(string),
`TASKPRE(string),
`TASKPOST(string)
to the functions and tasks where join points
need to be defined. They actually act as callbacks at the start and at the end of functions and tasks.

The `FUNCPRE(string),`FUNCPOST(string) callback macros are for join points in functions and the `TASKPRE(string),`TASKPOST(string) callback macros are for join points in tasks.

The differentiation is needed because SystemVerilog does not allow time-consuming instructions in functions.

THE FRAMEWORK IN ACTION

The diagram below compares the test case simulation flow with an object-oriented approach and using the framework.



Stage 0 is similar for both approaches.

Stage 1A assumes the extended classes which integrate the modified tasks and functions for the test case implementation, have been developed and compiled.

Stage 2A assumes from the verification engineer a clear view of the whole verification environment structure.

During the execution of stage 3A, reverting to base classes behavior is not a straightforward or flexible process.

Stage 1B assumes the framework has been integrated and the addAdviceFunction/Task methods implemented (if needed) for every class which require method modification for this specific test case.

During the execution of Stage 2B, reverting to base class behavior as well as changing the join points for a task or a function is a straightforward process.

CONTROLLABILITY APPLICATION

Below is a SystemVerilog test program for an environment (Environment) having a driver class (DriverClass) with a drivePacket method. The test first uses a modified drivePacket method, then at some point in the simulation, switches back to the basic drivePacket method:

```
program test;
Environment environment;
initial
begin
// drivePacket task of driver class is modified:
AOPAdvice#(DriverClass)::setAdvice(IS_ONLY,"drivePacket");
// Construct & run the environment:
environment = new(); environment.run();
…
// Original drivePacket task is restored:
 AOPAdvice#(DriverClass)::reset();
…
end
endprogram: test
```

CONCLUSION

TEST BENCH CONTROLLABILITY ASPECTS

Complex test cases involving modifications of some of the data classes in the generation part of the test bench and some of the architecture classes in the driving part of the test bench can be written in a more natural manner without extending a single class, letting the verification engineer focus on creating the sheer functionality sought from the test bench.

TEST BENCH OBSERVABILITY ASPECTS

Debugging the generation, driving or checking classes of the test bench with their various functional advices is efficiently achieved by using additional observation advices. Moreover, these observation advices allow for the visualization of dynamic objects alongside DUT signals, using SystemVerilog interface constructs and a waveform viewer. This is an appreciable acceleration compared to the standard console/file displays with reverse engineering methods.

GENERAL RESULT

The method advice feature of aspect-oriented programming is definitely a useful addition to standard object-oriented test benches, as shown by using this SystemVerilog framework, especially during the test cases development phase of a verification effort and for achieving robust test bench architectures by easily tracking down non-DUT related issues.

The SystemVerilog framework for adding the method advice feature of aspect-oriented programming for functions and tasks of the various classes in an object- oriented test bench is agnostic to the simulator used. Additionally, the reuse of the framework for subsequent projects is straightforward, as the advices added for a specific project do not interfere with the framework itself.

REFERENCES

[1] Robinson D., 2007, Aspect-Oriented Programming with the e Verification Language, Burlington MA, Morgan Kaufmann Publishers, Elsevier
[2] Hollander Y., Morley M., Noy A., 2000. The e language, a fresh separation of concerns. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=911754& tag=1
[3] Spear C., Tumbush G., 2012, SystemVerilog for Verification. 2012, springer.com
[4] http://en.wikipedia.org/wiki/Aspect-oriented_programming [5] Rosenberg, S., Meade K. A., 2010. A Practical Guide to Adopting the Universal Verification Methodology (UVM). San Jose, California: Cadence Design Systems, Inc.
[6] Cohen, B., Venkataramanan, S., Kumari, A., 2006, A pragmatic approach to VMM adoption … a SystemVerilog framework for TestBenches. Palos Verdes Peninsula, CA: VhdlCohen Publishing
[7] SystemVerilog Golden Reference Guide, Version 4.0, January 2006, Doulos Limited, Ringwood Hampshire

ACKNOWLEDGMENTS/CONTACTS

The author wishes to thank ARM Ltd. for providing the environment which enabled the development of the framework.

For any further details, please write to the following email address:
eric.ohana@arm.com