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

Eric Ohana

Media Processor Division

ARM Ltd.

Cambridge, United Kingdom

eric.ohana@arm.com

IEEE member – Circuits & Systems Society

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.

Keywords

Object-oriented Programming, Aspect-oriented Programming, Verilog SystemVerilog, ***e*** Verification Language, HVL, HDVL, Advice, Test Bench, Test Case, RTL, DUT, GPU, VMM, UVM, callback

Introduction

[3] explains that adopting a layered approach when developing a test bench yields a reusability feature when its building blocks are encapsulated.

The object-orientation features of SystemVerilog actually enable the adoption of a layered approach for constructing test benches.

There are two main types of classes in an object-oriented test bench:

1- Classes used to create stimuli to the DUT: the data classes. Objects created from data classes are dynamic in essence, in the sense that many of them are created and garbage collected during a typical simulation.

2- Classes used to create the structure used to convey the stimuli, to monitor, check and model the DUT activity: the architecture classes. Objects created from these classes are static in essence, in the sense that they are created only once and live throughout the course of a typical simulation.

Modifying the behavior of the methods in data and architecture classes is needed when creating different test cases, configuring the environment a DUT is instantiated in or for the debugging process. Those modifications are achieved through inheritance and polymorphism (method overriding) or also sometimes by simply changing the source code of the classes.

The first approach requires some level of object-oriented programming expertise in order to be accomplished efficiently and not create bugs in the test bench, which are hard to track down. It should also be noted that the specific purpose of a test case developed using this approach is not always as clear as it could be.

The second approach is not always possible, either because the original code is not modifiable for various reasons or desirable e.g. if the code is shared by several engineers, this could potentially cause a general stall in the verification effort, and creating many versions for a given class can render a verification environment prone to bugs.

A plain definition of the method advice feature of aspect-oriented programming is basically the possibility to modify the methods of a class without changing its source code and without the issues brought by inheritance and polymorphism. See [4] for a formal explanation. The paper shows that method advice can be particularly useful for verification purposes.

SystemVerilog does not currently offer inherent aspect-oriented features, and modifying the tasks and functions

using this HVL implies using the approaches aforementioned.
The paper describes a SystemVerilog framework. This framework enables the use of the aspect-oriented method advice feature, regardless of which simulator is used in the verification process.
The framework was used in a test bench during the course of a project at ARM Ltd, and appeared to be useful during the test cases development stages of the verification process. For the purpose of the paper, a simplified version of the test bench is used to allow focus on the method advice feature offered by the framework.

1- The SystemVerilog object-oriented test bench

This section gives a simplified overview of the test bench where the framework was used.
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.
Figure 1. below, 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.

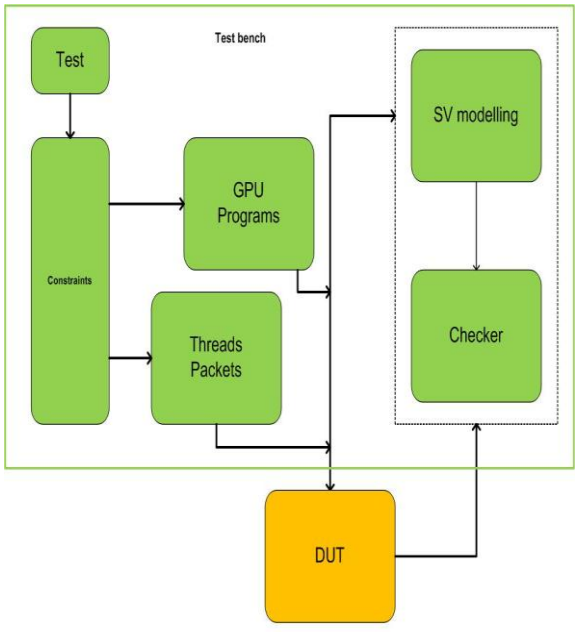

Figure 1. *The SystemVerilog object-oriented test bench*

Following is a non-exhaustive list of the type of randomizations the GPU programs and the threads packets undergo:

- The GPU programs exhibit specific sequences of instructions
- The threads packets are always longer than a given number
- The GPU programs use the minimum/maximum allowed allocation of dynamic work registers
- The threads packets are generated from a subset of all the possible types

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 ***Figure 1.***
- Introduce latency cycles on a control signal in an architecture driver class belonging to the **Threads Packets** block in ***Figure 1.***
- Displaying some debugging information when invoking methods in the classes of the **Checker** block in ***Figure 1.***

Note that for changing the randomization results of a data class, apart from inheritance and polymorphism (SystemVerilog constraint construct override) and changing the source code, it is also common to use the post_randomize() function.
The method advice framework consists of two parts: the first part is the library part and the second one is the verification environment part. After describing both part of the framework, comparisons between the object-oriented approach and the aspect-oriented one to achieve method modification are made considering two aspects for the verification environment: controllability & observability, whether conceptually or with specific examples.

2- The library part of the method advice framework

The library part of the framework consists of two SystemVerilog files:

1- An Aspect-oriented Programming Advice Package (AOPAdvicePackage.sv)

2- An Aspect-oriented Programming Advice utilities file (AOPAdviceUtils.svh)

The second part of the framework, which is the verification environment one, is a set of methodological rules on how to write a data/architecture class for using the method advice features. This is explained in the next section with specific examples and simulation outcomes.

The role of this framework's part is to manage a database reflecting the activity on the method join points. For the method join point definitions, see the AOPAdvicePackage.sv package description below. The database includes every method (SystemVerilog function or task) of every class in the verification environment.
This database concept is similar, in essence, to the UVM configuration database one. See [5] for more details.
The database is static in the object-oriented sense and can be modified at any point during a simulation.

The AOPAdvicePackage.sv package:

The package defines first 2 simple SystemVerilog new types:

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

2- ***typedef joinPointEnum joinPointQueue[$];***

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.

As for multiple method advices for ***IS_FIRST*** and ***IS_ALSO*** join points, it should be noted here that contrarily to ***e***, it is sufficient to define the join points (***IS_FIRST*** and ***IS_ALSO***) only once for the method of the class. The advice code itself is added in the verification environment part of the framework detailed in section 3.

joinPointQueue is a queue of ***joinPointEnum***.

The usage of these two new types is clarified with the remaining part of the AOPAdvicePackage.sv package.

The last part of the package consists of one parameterized class.

The parameterized class has one static property and four static methods.

The parameterized class:

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

The need for the class to be parameterized is explained in the paragraph explaining its static property below.

Its static property:

static joinPointQueue dataBase [string];

This string associative array of joinPointQueue keeps for every method name (the string argument) a queue of the join points set or cancelled for this specific method.

As the class is parameterized, an independent static database can then be accessed for every class type in the verification environment.

Its four static methods prototypes:

1- ***static function void setAdvice(input joinPointEnum joinPoint,input string methodName);***

This function sets a join point on a method name for a given class in the static property database: ***dataBase***.

Once a join point is defined on a method, it will be executed, in accordance to the overall definition of all the join points, until it is cancelled (see C-). The actual execution is explained with the second

Usage example:

AOPAdvice#(MyClass)::setAdvice(IS_ONLY,"myClassMethod");

2- ***static function bit getAdvice(input joinPointEnum joinPoint,input string method);***

This function returns 1'b1 if a join point on a method name for a given class in the static property database ***dataBase*** exists.

Usage example:

AOPAdvice#(MyClass)::getAdvice(IS_ONLY,"myClassMethod");

3- ***static function void reset(input string methodName);***

This function's role is to cancel all the join points defined for a method name of a given class. This is the way to resume original execution on a method.

Usage example:

AOPAdvice#(MyClass)::reset("myClassMethod");

4- ***static function void print();***

This function displays the method names and their join points for a given class, if any has been set and/or cancelled that is.

Usage example:

AOPAdvice#(MyClass)::print();

A typical output of the function call above would be:

Method(s) for class MyClassPackage::MyClass:

myClassDisplay -

IS_ONLY

post_randomize -

IS_ALSO

The AOPAdviceUtils.svh utilities file:

This file should be included in every class, which needs to use the method advice feature.

It consists of:

1- One external function and one external task for adding method advices.
2- Four macros:
   a. ***`FUNCPRE(string)***
   b. ***`FUNCPOST(string)***
   c. ***`TASKPRE(string)***
   d. ***`TASKPOST(string)***

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

3- The verification environment part of the method advice framework

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:

1- 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***. This definition occurs automatically by including the AOPAdviceUtils.svh file in the class definition.
2- 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 external function and the external task:

Below is the prototype for the external function:

```
extern function bit addAdviceFunction(input string methodName,input bit start = 1'b1);
```

Its content is basically the instantiation of a case statement which interrogates the method advice database for every possible join point existence for a function name and invokes a modified method call if true. It is typically defined in a class specific package, which includes the class itself and imports the AOPAdvicePackage.sv package described in the library part of the framework.

The case template is described in ***Template 1.***

```
case (functionName)
"myFunctionName":
begin
if (AOPAdvice#(MyClass)::getAdvice(IS_ONLY,methodName) && addAdviceFunctionStart)
begin // Modified function call occurs here return 1'b1; end
if (AOPAdvice#(MyClass)::getAdvice(IS_FIRST,methodName) && addAdviceFunctionStart)
begin // Modified function call occurs here return 1'b0; end
if (AOPAdvice#(MyClass)::getAdvice(IS_ALSO,methodName) && !addAdviceFunctionStart)
begin // Modified function call occurs here return 1'b0; end
return 1'b0;
end // case: "myFunctionName
default: return 1'b0;
endcase // case (functionName)
```

Template 1. *The case instantiation template for the* ***addAdviceFunction*** *external function*

Below is the prototype for the external task:

```
extern function addAdviceTask(input string methodName,ref bit addAdviceTaskStart = 1'b1);
```

The case template for the task is slightly different from the function one, as a task does not return any value in Systemverilog. It is described in ***Template 2.***

```
case (taskName)
"myTaskName":
begin
if (AOPAdvice#(MyClass)::getAdvice(IS_ONLY,methodName) && addAdviceTaskStart)
begin // Modified task call return; end
if (AOPAdvice#(MyClass)::getAdvice(IS_FIRST,methodName) && addAdviceTaskStart)
begin // Modified task call addAdviceTaskStart = 1'b0; return; end
if (AOPAdvice#(MyClass)::getAdvice(IS_ALSO,methodName) && ! addAdviceTaskStart)
begin // Modified task call end
addAdviceTaskstart = 1'b0;
end // case: "myTaskName
default: addAdviceTaskstart = 1'b0;
endcase // case (taskName)
```

Template 2. *The case instantiation template for the* ***addAdviceTask*** *external task*

The callback macros for the functions and tasks:

The callback macros are different for functions and tasks, as SystemVerilog does not allow time-consuming instructions in functions.

There are two places for instantiating the callback macros: one before the execution of the original content of the method (*PRE macros) and one after the execution of the original content of the method (*POST macros). Care should be taken to ensure the desired functionality is implemented when instantiating the *`FUNCPOST(string)* &*`TASKPOST(string)* macros: if the original methods return before reaching them, as they won't be executed.

Examples are described in ***Template 3.***

```
task myTask();
`TASKPRE("myTask"); // Handles IS_ONLY and IS_FIRST join points
// Here comes the original content of the task
`TASKPOST("myTask"); // Handles IS_ALSO join points
endtask: myTask

function myFunction();
`FUNCPRE("myFunction"); // Handles IS_ONLY and IS_FIRST join points
// Here comes the original content of the function
`FUNCPOST("myFunction"); // Handles IS_ALSO join point
endtask: myFunction
```

Template 3. *The callback additions for tasks and functions*

The callback macros validate the join points and call the actual modifications.

4- Controllability for the verification environment

Application to test case development

A test case development mainly involves one or both of the following actions on the verification environment:

1- Modifying the randomization parameters for the DUT stimuli data classes

2- Modifying the functionality of one or more of the test bench architecture classes' methods (task or functions. Examples of such classes could be driver classes, checker classes, monitor classes and so on.

Figure 2.compares the test case simulation flow with an object-oriented approach and using the framework.

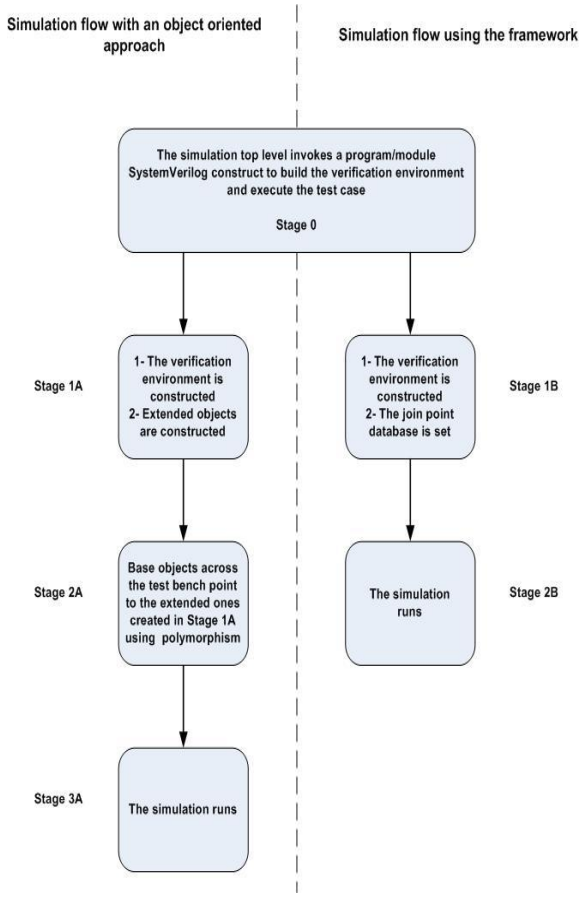

Figure 2. *Test case simulation flows compared*

Stage 0 of ***Figure 2.*** is similar for both approaches.

- Stage 1A of ***Figure 2.,*** 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 of ***Figure 2.***, reverting to base classes behavior is not a straightforward or flexible process.
- Stage 1B of ***Figure 2.,*** 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 of ***Figure 2.,*** reverting to base class behavior as well as changing the join points for a task or a function is a straightforward process.

Template 4. describes a test execution process in a SystemVerilog program construct, from the setting of the method advice database through environment construction and execution. In this specific example, the method advice database is reset in the middle of the test to cancel all join points and revert to basic method functionality.

```
program automatic test;
import AOPDefinitions::*;
import AOPAdvicePackage::*;
// Verification environment package:
import EnvironmentPackage::*;
// Verification environment handle:
Environment environment;
initial
begin
// Build join point database for test case …
// drivePacket task of driver class is modified:
AOPAdvice#(DriverClass)::setAdvice(IS_ONLY,"drivePacket");
// Construct & run the environment:
environment = new();
environment.run();
.
// Redefine database during test case …
// Original drivePacket task is needed:
AOPAdvice#(DriverClass)::reset();
.
end
endprogram: test
```

Template 4. *Test execution process with method advice*

The example in ***Template 4*** creates a test case by replacing completely the ***drivePacket*** task functionality with the ***IS_ONLY*** join point (e.g. the packets are driven to the DUT with higher latencies), this is achieved by the first underlined piece of code. At some point during the test, the verification engineer wants/needs to revert to the original functionality of the ***drivePacket*** task, this is achieved by the second underlined piece of code.

This example also highlights an advantage of the framework as the different method advices can be dynamically controlled at run-time, something which is not possible with the ***e*** Verification Language.

Also, the lines of code underlined in ***Template 4.*** are the lines to be modified when using an object-oriented approach: the extended class objects should be created at this stage, and then different class handles in the test bench should point to these new created objects, for example the following line from ***Template 4***:

AOPAdvice#(DriverClass)::setAdvice(IS_ONLY,"drivePacket");

Should be replaced by:

```
ExtendedDriverClass extendedDriverClass;
extendedDriverClass = new();
```

This assumes the existence of a new extended class (***ExtendedDriverClass extends DriverClass***) which includes an overriding method for ***drivePacket()***.

After the environment creation, all the handles pointing to a DriverClass object should be reassigned.

For one reassignment, that would be:

```
environment.<hierarchical path to the handle> .driverClass =
extendedDriverClass;
```

It is remarked here that providing join points to the SystemVerilog ***post_randomize()*** function of a data class is the current approach of the author to modifying a class randomization output. The SystemVerilog constraint constructs, although they can be overridden, are limited in terms of method call within the expression of the constraint itself. [7] details the exact features and limitations of SystemVerilog constraints.

5- Observability for the verification environment

Application to debugging

It is very useful during the debugging process of a verification issue, whether it is test bench related and/or DUT related, to be able to monitor specific parameters of the dynamic state of test bench objects along with DUT signal values. A VMM implementation of this concept is discussed in [6].

Template 5. is the procedural part of a systemverilog program construct where two join points on the ***checkPacket*** function are defined. The code in the ***addAdviceFunction*** function can for example calculate and print useful information to understand why a call of ***checkPacket(…)*** would returns an otherwise hard to explain error.

```
initial
begin
// Build join point database for debugging …
// checkPacket function of CheckerClass is modified:
AOPAdvice#(CheckerClass)::setAdvice(IS_FIRST,"checkPacket")
;
AOPAdvice#(CheckerClass)::setAdvice(IS_ALSO,"checkPacket")
;
// Construct & run the environment:
environment = new();
environment.run();
end
endprogram: test
```

Template 5. *Application to debugging example*

Moreover, using the technique shown in ***Template 5.***, the advices can be used to compute dynamic variables for various objects and then to print them or set them to SystemVerilog interfaces variables. The SystemVerilog interface variables can finally be viewed, along with DUT signals, on any waveform tool for debugging purposes.

Conclusion

Test bench controllability and separation of efforts 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 aspect

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.

Finally, it should be noted that using method advice for verification, whether it is for test cases development or debugging purposes appears to be a more natural approach from the verification engineer perspective.

It is mainly due to the fact that the approach relieves the engineer from caring about pure object-oriented issues like inheritance and polymorphism. It also allows her or him to focus on the sheer functionality sought from the test bench to achieve quality verification.

Reference

[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