Considerations for Development and Support of Exportable UVM IP

Steven K. Sherman

AMD
Boxborough, Massachusetts, USA

www.amd.com

ABSTRACT

UVM IP intended for export to other testbenches should carefully adhere to guidelines and best practices to avoid costly usage and maintenance issues. This paper discusses some of those considerations in practice within our verification group at AMD including special considerations with respect to adoption of UVM.

Table of Contents

1. Introduction 3
2. IP Export Challenges 3
3. Strategy 4
   Pre-processing *4*
   Switches *4*
   Parameters *4*
   Interconnect *5*
   Active/Passive *5*
   OVM vs. UVM Run Phases *6*
   Internal Checks ..... ***Error! Bookmark not defined.***
4. Development 6
   Directory Organization *6*
   Drain/Idle Checks *7*
   No Raised Objections *7*
   Verbosity and Messages *7*
   Message ID Uniqueness *8*
   Begin/End *8*
5. Checking 9
   Peer Review *9*
   Test Builds *9*
6. Delivery 9
   Pure UVM *9*
   Encapsulation *10*
   Imported IP *10*
   Avoid Compile Warnings *10*
   Black Box with IP Visibility *10*
   Configuration Database Object *10*
   Internal Variables *10*
   Analysis Port Connections *10*
7. Conclusions 11
8. Acknowledgements 11
9. References 11

Table of Figures

Figure 1 - Client Testbench, Passive Agent 5

Figure 2 - Client Testbench, Active Agent 6

1. Introduction

This paper is a qualitative survey of considerations for preparing IP written per UVM [1] [2] for use by others. As such, this paper is subjective and not comprehensive but leverages cumulative decades of verification experience among experts currently engaged in the process of generating and supporting exportable UVM IP within a large commercial enterprise, AMD. This paper suggests pragmatic approaches to address ill-defined, pervasive challenges in verification with respect to exporting UVM IP and with acknowledgement to a few recent works on the topic.

2. IP Export Challenges

After recent UVM training, some engineers felt (at first) that UVM does so much that perhaps verification will require little or no effort in the future. Experience indicates the challenges of the work to be done may be increasing faster than verification capabilities. It is more important than ever to deliver quality code on time and on budget. That includes exercising greater care in exporting IP to be used by others.

While it may be seemingly ideal to start developing UVM "from scratch," this still usually includes import of other UVM IP. Thus, while emphasis is upon exporting UVM IP, recognise that one who exports also imports. It makes sense to consider oneself as both importer and exporter of UVM IP.

Parallel to the import/export of UVM IP are respective specifications. Specifications often include details that are incomplete as well as details that are out of date. A team that supports exportable IP may face special challenges both with import of core technology and export of developed technology. Both import and export are moving targets, constantly upgrading with technology development as well as introduction of new products. Thus, there may seldom be such thing as a "golden spec" for verification purposes.

The dynamic role of importer/exporter with moving targets makes the tenets of UVM all the more critical in terms of keeping IP reusable, scalable and flexible. The challenge is multiplied with the need to support multiple clients. Major investment up front is necessary to avoid very costly debug and support after export.

Export of IP is usually with the intent that it be for "vertical reuse." Many pragmatic considerations have already been discussed in prior work. As discussed in [3], the tenets of UVM (and OVM) include functionality encapsulation, transaction-level modelling and more with emphasis on reuse. General considerations include ordering tasks, using template generators and scoreboards, code qualification (both automated and manual), incremental integration and so forth. With emphasis on vertical reuse in [4], further pragmatic considerations include careful consideration of active/passive configuration and connection, functional checks and coverage, messaging, test control, sequences, encapsulation and so forth. Use of parameterized interfaces [5] versus parameterized classes is a critical addition to the list of considerations.

This discussion builds upon these with a few additional pragmatic considerations as they relate to ensuring scalability, flexibility and reuse for exportable UVM IP. It is presented within the context of an iterative process that includes strategy, development and checking before delivery.

3. Strategy

A number of strategic considerations are discussed next, based on feedback from engineers and in no particular order.

Pre-processing

Use of some form of source code pre-processing can help ensure scalability at compile-time. That is, an automated method for generating UVM code can naturally encourage preservation of scalability. However, pre-processing should be simplified for exportable IP. Clients may not expect to provide extensive customization of the IP and will typically prefer a simple, straight-forward compile process. Ideally, the IP exported should require minimal or no pre-processing of the source. Thus, while pre-processing may be used for development, it should be simplified in the exported IP.

Switches

Any switches should be for features, not for specific client projects. This helps to ensure that switches make sense in the future long after particular projects have completed. Thus, the following would be unacceptable:

```
`ifdef PROJECT_X
```

The following would likely be acceptable:

```
`ifdef QUAD_CORE
```

Parameters

The testbench and RTL should be readily configurable by making changes to a command line (run-time) or to parameters within a configuration object (compile-time).

Hard-coded values at lower levels undermine scalability and cannot be completely avoided. However, use of parameterized interfaces (versus using parameterized classes) can help to minimize compile-time scalability issues due to hard-coded values [5]. This limits hard-coded values to interfaces and fits better with UVM's more object-oriented approach. During development, the hard coded values instanced at interfaces lower in the hierarchy should reasonably exceed the values anticipated for all iterations of the IP.

Caution should be exercised with respect to use of parameters in parameterized classes. If there are too many parameters then the use and maintenance of them can be unwieldy, undermining flexibility. More importantly, the parameters available should be limited to just those of value to the client.

Some argue that to avoid changes in the future everything should be parameterized. Given the need to limit the number of parameters, a balance should be struck between lower-level, hard-

coded parameters and parameters supported in interfaces. This is a subjective determination that must strike a balance between current client needs and future configurations of the IP.

There should be limited parameters at the top level of the exported IP with parameterization limited to interfaces. This can help avoid the overhead of supporting massive parameter lists that drop through several vertical layers. Parameterization should be pushed to lower-level models with the lowest level modules being highly parameterized, as a rule.

Interconnect

Use some form of automated method to generate interconnect between modules. Signal names should be well defined so that they may be used by all tools in the build flow. This can result in IP that integrates with minimal effort to compile and do initial reset testing.

Active/Passive

In theory, the IP developers should enable active components but export passive IP to clients who then supply new active components. In practice, the client may not have some active portions of the DUT ready but may still want to continue development. For this reason, it is crucial to ensure that the client be capable of using the exported IP active, then revert to passive when the active components are available. With this consideration, it should still be the default for the exported IP components to be in passive mode, requiring active mode to be explicitly ordered.

The figure below illustrates the intended configuration of the exported IP in the client testbench, using the client's active DUT:

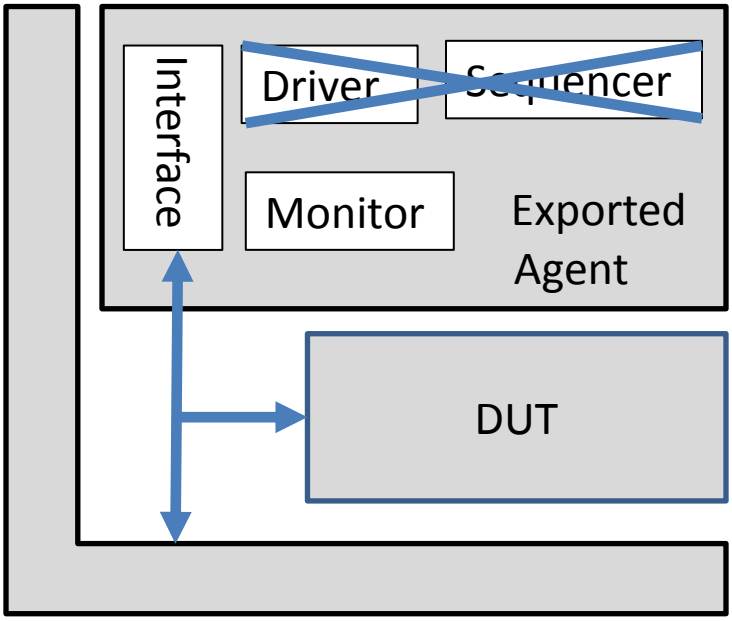

Figure 1 - Client Testbench, Passive Agent

The figure below illustrates the function of the exported IP active agent in lieu of the client providing the active DUT:

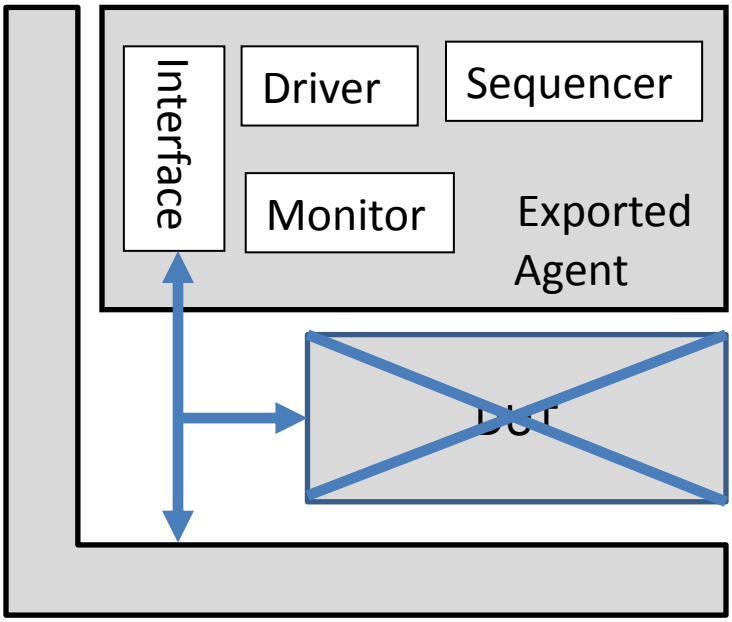

Figure 2 - Client Testbench, Active Agent

OVM vs. UVM Run Phases

It may be pragmatic to stick with the original OVM run phases and ignore the new UVM run phases. Customers that have not yet fully migrated to UVM may not support the new phases. Also, at this time the future of UVM phases appears to have some uncertainty.

4. Development

A number of development considerations are discussed next, based on feedback from engineers and in no particular order.

Directory Organization

The development directory hierarchy should separate the exportable IP from the testbench. Though both may be made available to the client, the general rule is that only the exportable IP is explicitly supported. This helps to avoid "code bloat" that can result from time-critical delivery of updated IP. Specifically, when there is a bug fix it is best to deliver just the IP required, not necessarily the complete testbench used to develop it or the underlying imported sub modules.

Though made available to the client, the testbench used to develop the exported IP should also be distinctly isolated from the code used by the client. The development testbench should be readily built and run by the client to allow for observation of the IP by the client outside of the client's testbench. In this way, the client may look to the testbench when looking for examples of how the IP should behave, when there are debug concerns and so forth.

Drain/Idle Checks

Perform drain/idle checks in the check phase. Specifically, development scoreboards should generate error if there are transactions missing at the end of simulation. However, upon export these errors should become switchable to warnings. It may, for example, be normal during client testing for some transactions to remain incomplete. So, a switch should be available to the client to set these warnings to errors.

No Raised Objections

Do not raise objections within the exportable code as this can interfere with client processes. Objections should only be raised in the testbench, whether it is that used in development or the client's testbench.

Verbosity and Messages

UVM verbosity practices should be implemented early in development to avoid later rewrite.

Ultimately, the exported IP should generate very few messages. By default, verbosity is set to UVM_MEDIUM. As a practice, most messages suitable for debug should be qualified by UVM_HIGH. A few critical, one-line messages, such as might indicate a component has started or stopped, should be qualified by UVM_LOW.

Use of some easily detected string such as "BOZO", "TODO" or "####" should be standardized among team members to tag questionable areas of code.

During development it is perfectly acceptable to make use of temporary UVM info messages with tags such as:

```
`uvm_info("####", "message", UVM_LOW);
`uvm_info("####", $sformatf("value = %0d", value), UVM_LOW);
```

Temporary messages should be deleted, commented out or reduced to a minimum before export. Messages deemed especially helpful can, of course, be set to UVM_HIGH and remain in the code. These can be enabled on a per-component basis from the command line with something like:

```
-vcs_opts "+uvm_set_verbosity=*.my_component,_ALL_,UVM_HIGH,run
"
```

Optionally, verbosity of other components can concurrently be reduced to a minimum with something like:

```
-vcs_opts "+UVM_VERBOSITY=UVM_NONE
+uvm_set_verbosity=*.my_component,_ALL_,UVM_HIGH,run "
```

Occasionally, it can be helpful to do special functions based on verbosity. This can be done by ensuring that components derive from the `uvm_report_object` rather than the more generic `uvm_object`. Then, one may check for local verbosity with something like:

```
full_verbosity = uvm_report_enabled(UVM_FULL);
```

In this example, `full_verbosity` could then be used to, for example, control special printing or other debug functions.

Message ID Uniqueness

Messages in exported IP should have unique ID strings. ID is based on whatever is put into the ID field of, for example, an instance of `uvm_info:

```
`uvm_info("ID", "Message", 0);
```

One approach to making such IDs unique may be to fill in this field with something derived using a variable:

```
`uvm_info($sformatf("ID:%0d", my_variable), "message", 0);
```

This should be avoided because the same message could then have multiple IDs, making debug confusing.

One might consider using a virtual function from a higher level that can prepend a string that includes hiearchy information.

```
`uvm_info(sig_id("SIGNATURE_TAG"), "message", 0);
```

From the higher level, this function might be defined with something like:

```
virtual function string sig_id(string sig_tag);
    return {"MY_COMP/NEXT_COMP_DOWN/", sig_tag};
endfunction
```

The client may then be able to customize this function, helping to also make the message unique across different instances of the same IP.

Begin/End

Consider use of "begin" and "end" in all constructs. This allows for ready insertion of UVM macros or temporary debug messages without fear of breaking the logic such as might have happened here:

```
if (flag)
   `uvm_info("####", "flag is asserted", 0);
   do_flag_action();
```

5. Checking

Checking is a combination of manual, human review combined with automated feedback.

Peer Review

A coding style must be centrally determined and enforced. There should be lots of coding examples available that comply with the accepted coding style. The actual coding style is subjective. It is more important that a single coding style be adopted so that any may readily inspect, comprehend and modify any module.

Both pseudo code and actual code should be independently reviewed both for content and for adherence to coding style. There should also be a process in place so that others can apply fixes that can be tracked, reviewed and regressed. This is especially critical as designers migrate to UVM.

With UVM many component instances are readily generated, even if they are not really needed or used. Thus, from time to time audit the instances in the design. Look for instances of modules being created that are not needed (and may be slowing the system down). For example, turn off or delete unused interfaces, model instances, queues and so forth.

Test Builds

As soon as feasible, exported IP should be built into all available testbenches at all levels of hierarchy. This will help to expose issues not visible in the development testbench early when they are relatively easy to address.

6. Delivery

Here are guidelines for the exported IP deliverables, organized from general and higher level to more specific and low level. Note that we do not bother with encryption as our customers are internal only.

Pure UVM

Conversion of imported OVM to UVM may be necessary, but only pure UVM should be exported.

Though the exported IP should be pure UVM, do not expect the client to abide by the same restriction. Similarly, do not expect the client to adhere to any one style of documentation. Simply strive to make the documentation for the exported IP as easy as possible for the client to access and as UVM-pure as possible. Let the client determine their own standards beyond these.

Though comprehensive documentation of the exported IP may be optional, a client will expect adherence to specifications and timely updates of the exported IP in response to updated specifications. The client will also expect some form of documented user guidance that highlights important details.

Encapsulation

A client should expect a package file that brings in the exported IP. The documentation should include a guide to how to configure the IP along with local configuration and bind files. This is in support of "encapsulation" of the exported IP [4]. This should not require the user to have in-depth knowledge of the IP but the user should be free to peruse all sources.

Imported IP

The exported code should also include any IP required by the exported IP and imported by the developers. Hooks should be available to enable the top-level testbench to connect to alternative IP imported by the client. The respective active components for imported IP for those do not need to be made available to the client outside of an example testbench for the exported IP.

Avoid Compile Warnings

Ideally, the delivered code should build without VCS compile warnings. In general, clients should be free to promote warnings to errors without having to modify the exported IP. Only a handful of warnings should be accepted with the understanding that these should never be promoted to errors.

Black Box with IP Visibility

A client will generally prefer to regard the IP as "black box." But, the client will also demand visibility into the IP sources. Though an IP update may be timely, the client may justifiably not want to wait and should be free to make local modifications and build with those changes pending a more formal release.

Configuration Database Object

A UVM configuration database object should be included and well-documented. This should enable the client to readily disable any checkers or scoreboards, control specific component verbosity and have access to other configuration options.

Internal Variables

Somewhat contrary to object-oriented theory, the user will likely demand access to a limited number of internal variables for debug and development purposes. Thus, the guidance should clearly indicate which such variables are accessible, usually through function calls. The guidance should not include access to variables that will be of little interest to the user, which should include the majority of variables.

Analysis Port Connections

UVM automatically manages analysis port connections. But, this can also be a hindrance to clients as they debug. Specifically, while debugging, a user may unsuccessfully search the code for the source of a "`write`" function. While the analysis port write functions may be found, a search of the source code may find no calls to any matching functions! To avoid confusion, ensure that all IP analysis ports are documented for the client. With this, the client can anticipate that a "`write_*`" is being initiated from a particular analysis port. This should include a description of the anticipated name for the port source ("`*_ap`") and potential destinations ("`*_xp`").

7. Conclusions

This paper has presented a subjective, qualitative collection of pragmatic considerations for UVM IP export. These considerations extend ideas presented previously, serving as further suggestions for teams engaged in development of UVM IP for vertical reuse.

8. Acknowledgements

Special thanks go to fellow AMD team verification engineers Joe Cruz, Thomas Loftus and Brian Rost for developing and contributing to the concepts presented in this paper.

9. References

[1] VIP Technical Committee. *Universal Verification Methodology (UVM) 1.1 Class Reference.* Accellera. June 2011 <http://www.accellera.org/downloads/standards/uvm/UVM_1.1_Class_Reference_Final_06062011.pdf>.

[2] VIP Technical Committee. *Universal Verification Methodology (UVM) 1.1 User's Guide.* Accellera. May 2012 <http://www.accellera.org/downloads/standards/uvm/uvm_users_guide_1.1.pdf>.

[3] El-Metwally, Moataz. "Guidelines for Successful SoC Verification in OVM/UVM." Design & Reuse. May 2011 <http://www.design-reuse.com/articles/26341/guidelines-for-successful-soc-verification-in-ovm-uvm.html>

[4] Litterick, Mark. "Pragmatic Verification Reuse in a Vertical World." DVCon 2013. Feb. 2013

[5] Yun, Wayne. Zhang, Shihua. "Deploying Parameterized Interface with UVM." DVCon 2013. Feb. 2013