Tweak-Free Reuse Using OVM

Sharon Rosenberg
Cadence Design Systems, Inc.
2655 Seely Avenue
Telephone #, 1-408-9146341
sharonr@cadence.com

ABSTRACT

Most companies today aim to leverage existing design and verification IP as part of the design and verification flow. Internal IP is developed, tuned and reused over time to become a major company asset and can be a competitive differentiator. A key requirement for developing a central verification IP (VIP) repository is to avoid the need to understand the implementation details or modify existing IP for use in follow-on projects. In working with many large and small corporations, we find that while many companies strive for such cross-company (and cross-project) component reuse, only a few manage to achieve this goal. This document describes the recurring practices that allow companies to excel in productivity and reuse.

1. INTRODUCTION

Unlike design, where specifications can (theoretically ☺) capture the desired functionality in a complete and deterministic form, the verification process is fluid, dynamic and unpredictable. A verification environment architect cannot foresee each and every corner case test for a certain project, or future projects that may involve reuse as you move between block and system level (vertical reuse) or reuse between projects (horizontal reuse). Success in this area involves a thought-out methodology that includes up-front planning, consistant organization combined with the ability to react to unplanned design changes or esoteric test requirements. The OVM has proven to be an ideal platform both for dealing with these issues and for implementing reusable verification components. This paper covers the essence of OVM and the unique characteristics that enable both the initial correct construction of reusable verification components and the ability to adopt changes to them for unseen requirements.

2. THE BIG STORY OF OVM

OVM is a multi-language methodology for the efficient creation of reusable testbenches (SV, e, SC). It is based on the proven foundations of eRM, factors in multiple perspectives and expertise from multiple vendors (mainly Cadence and Mentor), and is tuned to user's needs (OVM Advisory Group). For two years, OVM-SV has demonstrated huge success and momentum within the industry. There have been thousands of downloads from ovmworld.org, adoption and standardization by many corporations, active users and a massive user-community that has already developed much more commercial and internal verification IP than with any other methodology. What is unique about OVM is that it is a single, well-designed methodology. Like other methodologies, it provides a base class library and automation, (OVM automation is very capable, greatly appreciated and constantly adopted by both users and other commercial methodology developers ☺), but it also provides a high-level concept and a recipe for building **reusable** components. This high-level methodology is the class library backbone, and the power automation serves this high-level goal. The companies that have been successful in deploying a verification methodology understand that while some features are useful and should be used for easy coding and maintenance, other features are **critical** to reaping the benefits of the OVM. For example, you may choose not to use the synchronization classes (ovm_event_pool, ovm_barrier_pool) and implement a different facility to achieve this goal. If you decide not to adhere to the consistent OVM topology, you will be facing more severe implications. Independent islands of users and teams will result in different verification components that may be limited to their initial development needs, harder to read and understand by others, and prevent co-existence and further reuse. The rest of this document is dedicated to the high-level concepts of OVM that put you in the same league as the top-performing verification teams.

3. OVM STANDARD TOPOLOGY AND HIGH-LEVEL ENCAPSULATION

Object-oriented programming methodology calls for class-level reuse, meaning that each class has a well-defined signature of external services it provides and an internal implementation that should not externally accessible. This contrast between the server class and the client classes allows distributed development and maintenance of classes. As in a software development project, construction of verification components also benefits from this separation. However, verification environments have much more in common between themselves than with generic software development. Verification environments all need one component for generating traffic, one for driving a data item into the design and another passive component for monitoring the bus and doing checks and coverage. For a specific interface, a set of such component classes needs to be instantiated and connected, then share common configuration information. For some protocols, layering is needed, and other joint resources need to be shared. This requires both understanding and effort from the environment integrator. OVM provides this for you using a well-defined topology with high-level encapsulation, which means the environment developer only needs to to instantiate and pre-connect the classes in a standard manner within a larger container. Standard configuration attributes and a mechanism for controlling the reusable cluster of classes minimizes the effort and understanding required. Figure 1 describes the first level of containment in an "agent" container. A standard OVM agent has an active operation mode in which traffic is generated and injected and a passive mode in which only coverage and checking is performed. Passive agents are instantiated to monitor RTL devices and the standard configuration switch allows vertical reuse as external interfaces become internal in larger systems and there is a need to stop traffic injection. For some protocols, multiple agents are needed per interface. Per application, they need to share the same configuration, virtual interfaces, common resources(monitor), etc. OVM standard recommends a reusable OVM verification component (OVC). (See Figure 2) Changing the number of agents and the

environment configuration is achieved using the standard configuration mechanism. Following the high-level encapsulation is key for both environment developers and users. Developers who follow these guidelines adopt a correct basic topology, preventing thoughts such as “what was he thinking?” when an exotic topology is enthusically described within a code review. For users, adopting a new verification IP is simplified and the environment hierarchy is clear for both commercial and internal verification IP.

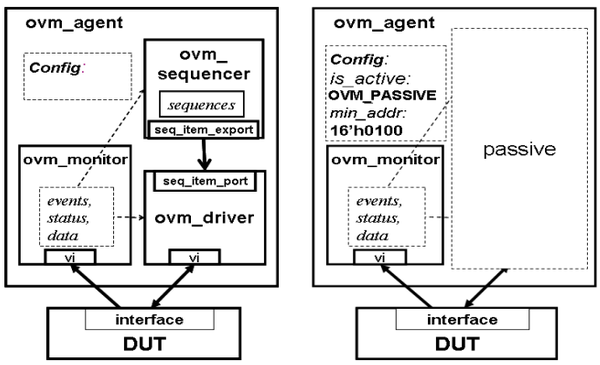

Figure1. Standard OVM Agent in Active and Passive Modes

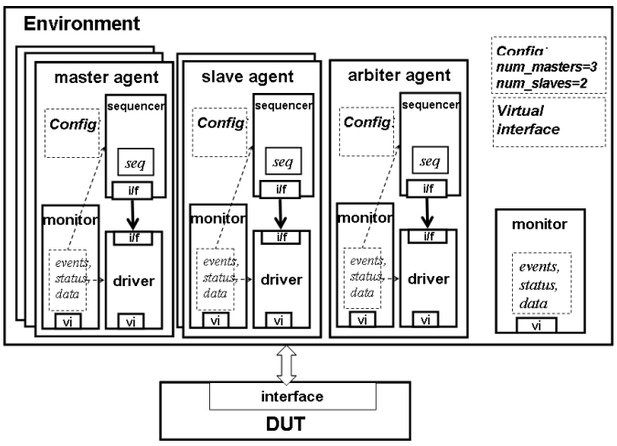

Figure 2. Standard OVM Environment

Figure 3 illustrates complete testbench integration. Notice that the testbench instantiates reusable verification environments (OVCs), as opposed to agents directly. This reduces the instantiation and configuration effort and makes the environment consistent for all verification IP. It is recommended to avoid short-cuts such as instantiating agents directly in testbenches or not differentiating between an agent in active and passive modes. So **Rule #1** for tweak-free reuse: **Use the standard OVM topology and configuration attributes.**

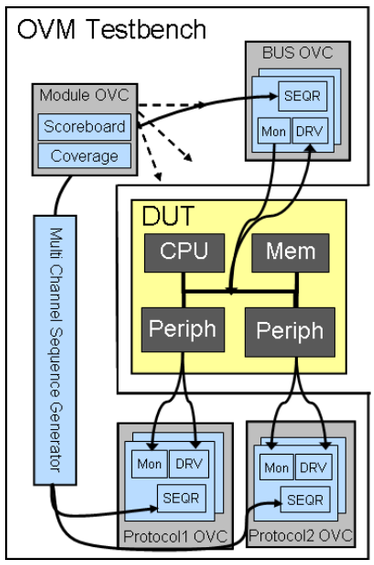

Figure 3: Standard OVM Testbench

4. CODE GENERATORS

Some may think “Isn’t using the clean OVM definition introduces overhead? Also there is much more rules to learn right?” After reviewing many OVM adoptions, users typically experience significant speedup in both development and integration of standard OVM environments. In addition, the standard hierarchy allows further automation as more commonality is defined. Code generators can leverage the canonical structure and produce bug-free, nicely commented code, with ever more consistency between components. The Incisive Verification Builder (IVB) is a wizard technology that produces OVM standard environments. The user provides input about the desired verification IP environment. Based on the initial input more questions are imposed untill the system is ready to produce a verification environment skeleton. A set of finalization action items is provided to guide users on how to complete the protocol specific OVC environment. The IVB technology has benefits both to naïve users (no need to know all the guidelines and rules) and experts alike (productivity tool, “why should I start from scratch?”). **Rule # 2** for tweak-free reuse is what many managers ask their team to do: **Start your reusable environment development from a proven code generator.**

5. PACKAGING AND NAME SPACES

Name space collisions are the shortest path to uninvited code modifications. In order to prevent name collisions, it is imperative to use the language name-spaces constructs in ***e*** or the package construct in SystemVerilog. But the packaging definition in this paper goes beyond keeping your global name-space clean. Standard directory structure, a designated location for the documentation, consistent file names, appropriate “include” or “import” schemes, and version control facilities are all examples for packaging guidelines that seems insignificant -- but have shown huge impacts on teams’ ability to share and maintain code. While packaging requirements should be intuitive and enable reuse, it is more important to adhere to a common set of packaging guidelines. Experience shows that various companies have different views on

the optimal guidelines. It is critical to fight the temptation to "improve" the standard directory structure ☺. Cadence OVM-ML contribution contains a de-facto standard directory structure and packaging guidelines that were already adopted by many users and commercial VIP providers within the last seven years. This gets us to **rule #3**: **keep your global name space clean and adhere to the OVM packaging guidelines**.

6. TEST CREATION AND RANDOMNESS

In Coverage Driven Verification methodologies, the user creates a smart testbench that can randomly create legal stimuli. Tests are layered on top of the testbench to target coverage holes. While the concept is simple, a flexible solution is needed to allow steering the smart machine without major re-writes. For example, one technique for nailing a corner case is reactive generation where traffic is generated taking the design state into account. Or users may want to use both procedural style to dictate order of transactions, or declarative constraints to alter the randomization (this can be achieved by the OVM factory). Using a randomization scheme that does not accommodate broad variety of test requirements may yield expensive re-write of the randomization scheme. OVM introduces the sequences mechanism that answers multiple necessities that test scenarios impose in a reusable way. Sequences are a set of interesting data-items that are tagged with a name, and can be reused later by tests and by other sequences. Sequences can be parameterized (for example a configuration sequence can include an enumerated type field that allow the user to select the type of initial configuration via constraint). OVM Sequences include much functionality and built-in solutions that the user may not know or care about in his implementation code. However as coverage goals compel an unplanned randomization, they are flexible enough to serve the new need. And with this we have **rule #4: Use the OVM sequences**

7. EXTENSIBILITY

As was discussed in the introduction, in verification you can and should plan as much as possible to accommodate wide-range of scenarios. However, when you build generic components you can never predict the unique configuration and traffic that will be necessary to place a DUT in a certain state. In some cases, users do not fully comply with the protocol or optimize it for a certain project. The ***e*** language, with it's Aspect Oriented Programming (AOP) nature, was brilliantly designed with the "design for a change" concept in mind. Every construct of the language can be modified or further characterized without touching the original source code. For example, you can add more struct members (constraints, coverage definitions, assertions etc.) to a packet definition without deriving a new type and introducing it to a system. OVM suggest two SW design pattern solutions to enable extensions: Callbacks and type safe factory.

7.1 CALLBACKS

Callbacks are pre-determined strategic points in time in which the reusable environment developer allows users to introduce their own procedural extensions. A user can attach callback classes to various components and define his extension logic to be executed by the reusable environment. Callbacks from multiple originators can be combined and ordered into a single joint environment. The major drawback of callbacks is that the developer needs to predict the callback location in advance -- in contradiction with our initial assumption that such predictions can not be made. Other disadvantages of callbacks include risks for hard-to-debug spaghetti code (as opposed to structural enhancements), hard-to-add declarative class members (can add these to the callback class but need to provide much context to it), and the risk of external interference with callbacks that should be executed in a certain order.

7.2 OVM FACTORY

The OVM factory is an advanced implementation of the classic software design pattern that is used to create generic code, deferring to run-time to decide the exact sub-type of the object that will be allocated. In functional verification, introducing class variations is frequently needed. For example, in many tests you might want to derive from the generic data item definition and add more constraints or fields to it. You might want to use the new derived class in the entire environment or only in a single interface. Both infrastructure components and data-items can be allocated via the factory. For example you can modify the way data is sent to the DUT by deriving a new driver. The advantage of factory over callbacks is that every polymorphic construct can be extended. You can prepend, append and override virtual tasks; you can add class members -- all in a structural easy-to-debug manner. The disadvantage of a factory is the lack of ability to combine orthogonal extensions.
And here is **rule #5** (for OVM-SV only): **Allocate components and data-items using the OVM built-in factory.**

8. MESSAGING REQUIREMENTS

The rules in this paper discuss important concepts such as architecture, randomness, extensibility, etc. One important topic that can force code modifications is trace messages. The main issue with the good old $display is the lack of ability to control the printed message from outside (without original source code modifications or re-compilation). Whether or not you use facilities such as directing output to a file, change the format and much more, surely you will find yourself enabling and disabling trace messages for areas that are suspected to be malfunctioning. The OVM report mechanism provides advanced message services. With the reporting mechanism is it important to use the macro messages as opposed to direct ovm_report_* calls. This is the only way to avoid expensive redundant string manipulation. The message macros contributed by Cadence were added to the joint release of OVM2.0.3. Another key feature to enable message consistency is related to the field automation capabilities that are also macro enabled. Asking the users to manually implement the do_print() functionality is not only extra development and maintenance effort, but it is a sure path for inconsistency in integration. When the integrator picks-up environments that originated from different resources, they are still interested in a coherent log files from their joint testbenches. They also expect format directives (such as print_options) to be served. I know that we all trained to despy macro usage at all cost, and we understand macro limitations. However, the main differences that we see out there between the field automation macro users and the ones that preach against it, is that those that recommend not using it have never tried to do so (☺). The field macro automations are simple, they work, are easy to maintain and rarely require debug. You can read more about it in forums and from users that were curious enough to try them out.
So write this one down as Rule number **rule #6: use a messaging facility**

9. OBJECTIONS AND END-OF-TEST MECHANISM

In simulation, agents may have a meaningful agenda to be achieved before the test goals can be declared as done. For example, a master agent may need to complete all its read and write operations before

the simulation should stop. A re-active slave agent may not object end-of-test as it is merely serving requests as they appear without a well-defined agenda. Only once all the components that raised end-of-tests objections drop them, the simulation can stop. The first objection mechanism was introduced in eRM 1.0 (2002) to facilitate this need. The objection mechanism is hierarchical and allows containers to own their sub-components objections, add drain-time or perform any other operation before propagating the objection up the hierarchy. This feature is needed for vertical reuse as systems that manage their own objections can be combined into a larger system. Without an agreed-upon end-of-test mechanism, the integrator would have to invent a mechanism to synchronize between, multiple end of test solutions and may have to modify the implementation of a reusable component that unliterary forces end-of-test.
Rule #7: Use the OVM end of test mechanism

10. MULTI-LANGUAGE (ML) SOLUTIONS

Multi-language design and verification is not a goal by-itself; it is more a fact-of-life and an opportunity. No one chooses to build a multi-language testbench, but they surely want to leverage all verification assets without re-writes (which is an extreme case of a tweak). However, users choose a methodology. Whether you already have multiple internal VIPs implemented in multiple languages or whether you will run into the requirements in the future, you probably need a multi-language methodology. The TLM is a multi-language (e, SV, SC) standard that facilitates transaction level communication. It reduces the need to learn and bridge between facilities with different semantics. However, some think that all you need for multi-language simulation is the TLM functionality and a free weekend, but TLM is just the basics of ML operability. Central configuration mechanism, traffic randomization and coordination, messages and other facilities are needs for practical multi-language simulation. OVM uses TLM for all standard languages and examples for multi-language usage for both TLM1 and TLM2 is demonstrated in the Cadence OVM contribution and Cadence releases**. Rule #8: Choose a multi-language methodology**

11. COMPLIANCE CHECKLIST

Code generators are efficient productivity tools. But if you did not buy a commercial VIP, you still need to go through the protocol specifications and complete the wished for OVC. While template-driven solutions can start the verification engineer on the right path, he can still stray from the methodology. It is the OVM Compliance Checklist, and associated automatic checkers, that can assure the author builds and the integrator receives OVCs compliant to the methodology, leading to tweak-free reuse.

The OVM compliance checklist contains the following categories:

- Packaging and Name Space – guidelines on how to package environments in a consistent way for easy shipping and delivery.
- Architecture – Checks to ensure similar high-level topology for OVM environments. This is critical for understanding a new OVC, its configuration and class hierarchy.
- Reset and Clock – various reset and clock topics
- Checking – touches the self checking aspects of reusable OVC
- Sequences – practices on creating reusable sequence library and correct setup
- Messaging – defines message methodology to allow user to efficiently debug environment and reduce support requirements
- Documentation – captures the requirement for a complete documentation requirements
- General Deliverables – more delivery requirements
- End of Tests – minimal end-of-test requirements
- OVM SV specific compliance checks – checks that are specific to OVM SV implementation.

Rule #9 for tweak-free reuse: **Use the ovmworld.org OVM compliance checklist**

12. AUTOMATIC COMPLIANCE CHECKING

Many customers who have been using the OVM and applying the compliance checklist have requested an automated tool for checking VIP against the checklist. AMIQ, an EDA company that is part of the growing OVM ecosystem, has enhanced a tool called “DVT” to automate OVM compliance checking. DVT is an Eclipse-based integrated development environment (IDE) targeted at increasing productivity of OVM SystemVerilog and e developers.
DVT also provides a broad range of OVM compliance review features based on the OVM Compliance Checklist, including:
- Overview of the environment architecture
- Customizable checks (grouped in categories including architecture, stimuli, checking, coverage, messaging, reset handling, packaging, etc.)
- Statistics (sequence library, checks, coverage groups, etc.)
- Graphical user interface (filters, search, etc.)
- Direct jump to problematic source code
- Integrated review and development (checks are refreshed as you fix errors)

After performing the analysis and adjusting the reusable environment accordingly, the VIP developer can export an OVM compliance HTML report to be delivered with the verification IP.
For more information on DVT please refer to http://www.dvteclipse.com

And the final rule for today is **rule #10: Use an automatic compliance checker**

13. SUMMARY

As the title promises OVM enables a way to avoid tweaking reusable code. You probably noticed that it is also a powerful catalyst for ease-of-use and productivity. There are lots of contradicting verification methodology recommendations available in books and blogs. This paper outlines a set of practical proven guidelines (as opposed to lab level recommendations) for developing reusable verification components. The important thing to remember is to follow the user manual recommended methodology and don’t take shortcuts. On December 23, 2009, the Accellera Verification technical sub-committee voted that the OVM and it’s source-code will be the basis for an industry standard library. This is great news for the SV user communities that were fragmented between commercial and home grown methodologies, and it’s also another reason to embrace OVM. Exciting times are a head!