Open Verification Methodology

Open Verification Methodology (OVM)

Built on the success of the Advanced Verification Methodology (AVM) from Mentor Graphics and the Universal Reuse Methodology (URM) from Cadence, the OVM brings the combined power of these two leading companies together to deliver on the promise of SystemVerilog.

The Open Verification Methodology (OVM), a joint development initiative between Mentor Graphics® and Cadence® Design Systems, provides the first open, interoperable, SystemVerilog verification methodology in the industry. The OVM provides a library of base classes that allow users to create modular, reusable verification environments in which components talk to each other via standard transaction-level modeling (TLM) interfaces. It also enables intra- and inter-company reuse through a common methodology with classes for developing stimulus sequences and block-to-system reuse.

Supported on multiple verification platforms, the OVM is the de facto standard methodology, ideally suited to speed verification for both novice and expert verification engineers. Built on the success of the Advanced Verification Methodology (AVM) from Mentor Graphics and the Universal Reuse Methodology (URM) from Cadence, the OVM brings the combined power of these two leading companies together to deliver on the promise of SystemVerilog. The OVM offers established interoperability mechanisms for verification IP (VIP), transaction-level and RTL models, and integration with other languages commonly used in production flows.

The Promise of SystemVerilog

Engineers designing and verifying complex electronic devices love standards. Standard languages, libraries, interchange formats, interfaces, and connectors make it much easier for design representations, EDA tools, and final products to interoperate. Thus, when a new standard comes along, there is a great deal of excitement and anticipation for the benefits engineers will accrue by adoption.

The SystemVerilog language is one such standard. After several years of intensive development work, it was ratified by the IEEE as Std. 1800-2005. As the industry's first hardware design and verification language (HDVL), SystemVerilog held out the promise of being a single expressive format widely adopted by engineers and supported by all manner of EDA tools. This would allow designs, models and verification components to be easily moved from one tool to another.

In truth, a language alone is not enough to define verification interoperability. Tool portability is critical, but beyond that it's necessary for verification components to follow a common methodology. Language features are used to assemble a set of building blocks that can be leveraged to create a sophisticated testbench or verification environment. In the object-oriented world of SystemVerilog verification, these building blocks are class libraries. A methodology with examples then shows users how to leverage the libraries to build reusable verification environments. Figure 1 shows the typical hierarchy relating the language and libraries to the simulator that runs them.

While all the major EDA vendors adopted an approach similar to Figure 1, the details differed. Each vendor had a different methodology, a different class library, and different language features used in its library and recommended by its methodology. Although the methodologies were internally consistent, they did not work with each other, thus compromising the promise of SystemVerilog interoperability. In addition, the fact that different simulators supported different subsets of the language, coupled with the fact that some methodologies were proprietary and restricted, meant that it was not possible to run VIP and verification code on multiple simulators.

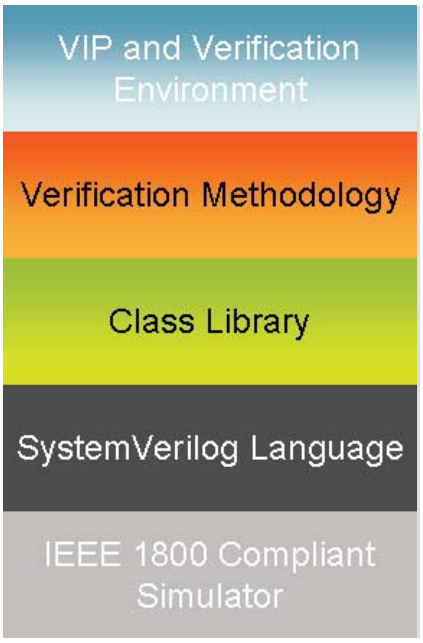

Figure 1: SystemVerilog Verification Hierarchy

Even assuming that all major simulators would eventually support the same set of language features, the existence of multiple class libraries and methodologies meant that VIP was not interoperable. Since the different methodologies defined different mechanisms for communicating between VIP and testbenches, combining components from different vendors was such a significant challenge that it offset the advantage of licensing pre-verified VIP in the first place. Designers and verification engineers were not seeing the benefit they expected from SystemVerilog adoption and urged the EDA vendors to address the situation.

A Truly Open SystemVerilog Methodology

In response, Cadence and Mentor Graphics developed a common methodology and a class library that runs on simulators from both companies. Announced on August 16, 2007, the OVM spans the class library and methodology layers of the SystemVerilog verification hierarchy, as shown in Figure 2. The class library is supported by both the Mentor Graphics

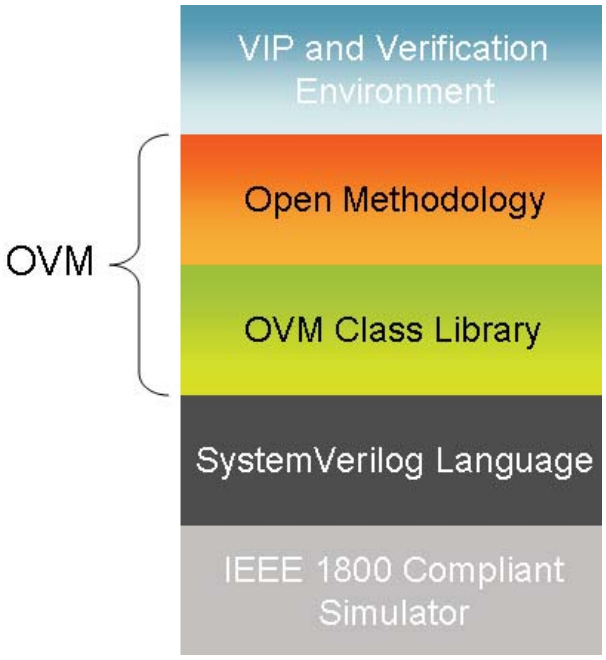

Figure 2: OVM Verification Hierarchy

Questa® and Cadence Incisive® verification platforms. Thus, any VIP or testbenches built using this library will run on either platform with no conversions, translations, or extra effort required. In fact, since the OVM is delivered in open-source format, the code will run on any simulator that supports the SystemVerilog standard.

As previously noted, verification interoperability requires more than common language support. Because of the alignment on class library and methodology, VIP available from Cadence, Mentor, or their partners will support the OVM and will interoperate seamlessly with testbenches also developed using the OVM and its library. This is an enormous benefit for both VIP providers and verification engineers developing testbenches. Under the OVM, communication among VIP and testbench components uses a SystemVerilog implementation of the widely adopted transaction-level modeling (TLM) standard originally developed by the Open SystemC Initiative (OSCI). In addition to fostering interoperability among SystemVerilog components, the choice of TLM makes it easy to integrate verification components written in other languages, such as *e* and SystemC, and to enable reuse of verification components, models, and environments at multiple levels of abstraction—from the block up to the system level.

The SystemVerilog OVM class library source code, usage examples, and supporting documentation are now available on OVM World at www.ovmworld.org. This is a completely open Web site, placing no restrictions on who can access its contents. The OVM class library source code is being released under the terms of the Apache 2.0 license agreement, a widely used open-source agreement that stipulates little more than retention of copyright notices. Customers, partners, standards organizations, and even EDA competitors of Mentor and Cadence can download the source code and use the OVM without having any monetary obligations to either company.

The object-oriented nature of the OVM and its class library means that most users will be able to extend its functionality without having to modify the source code. However, the Apache 2.0 license allows modifications or derivations if so desired. User community input will be solicited on the Web site to help the OVM evolve with additional functionality in the future.

Built On Proven Verification Technology

The OVM is "new" in the sense that it was recently announced. However, the OVM is already proven because it is built on well-established verification technologies and methodologies, reflecting more than ten years of industry best practices. Specifically, the OVM is based on, and backward compatible with, the Mentor Graphics AVM 3.0 and Cadence URM 6.2 versions.

The AVM was first announced by Mentor in 2004. Since then it has been widely adopted by many verification teams. AVM 3.0, available since May 2006, is the third generation of the methodology, adding new functionality and reflecting considerable input from the user community. Verification engineers using AVM 3.0 will be able to migrate quickly and easily to the OVM, providing portability and interoperability.

The URM was introduced by Cadence in 2006, expanding to encompass SystemVerilog support for the object-oriented verification techniques available since 2002 in the *e* Reuse Methodology (eRM). Verification engineers and VIP partners using the URM 6.2 release will be able to convert seamlessly to OVM compliance.

The rapid availability and robustness of the OVM is due to the complementary nature of the two predecessor methodologies, both of which were based on TLM communication and closely aligned in terms of methodology and functionality. The actual implementation of the OVM is a truly collaborative effort by both companies to provide the features required of such a state-of-the-art methodology. The collaboration involved a great deal of "give-and-take" from both companies, combining the best features, usability, and SystemVerilog knowledge garnered from many years of experience across the joint development team.

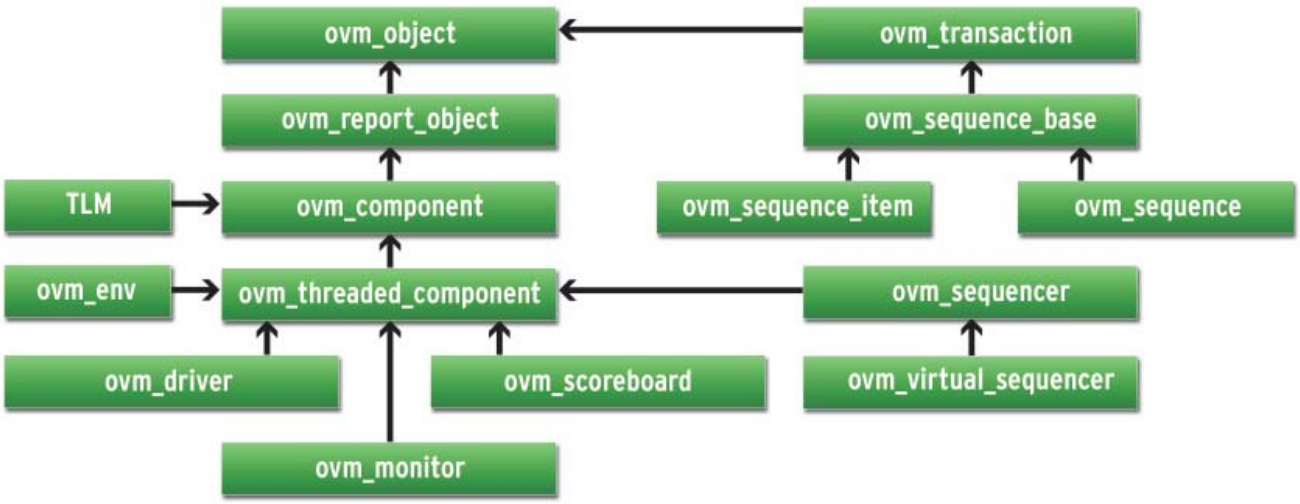

Figure 3: The OVM Class Library

The OVM Library

Figure 3 is a Unified Modeling Language (UML) diagram of the OVM library. The library and methodology provide all the technology needed to construct reusable constrained-random, coverage-driven testbenches. This technology enables an unprecedented level of flexibility, customization, and reuse.

The OVM provides the TLM-based infrastructure for building modular, reusable verification components that communicate through well-defined transaction-level interfaces. Its class library allows users to create sequential constrained-random stimulus, collect and analyze functional coverage information, and include assertions as first-class members of the configurable testbench environment. Specific features include the following.

- TLM communication as the underlying foundation for connecting verification components to facilitate modularity and reuse
- Common user-extensible phasing to coordinate execution activity of all components in the environment
- Ability to modify testbench environments on-the-fly and write multiple tests from the same base environment with minimal code changes
- Common configuration interface, so all components may be customized on a per-type or per-instance basis without changing the underlying code
- Straightforward test writer interface for configuring a testbench and specifying constrained layered sequential stimulus
- Common message reporting and formatting interface

Feature Overview

TLM Communication

OVM components communicate via standard TLM interfaces, which improve reusability. TLM defines a standard set of interface methods to define communication semantics but separates the interfaces from their implementations. Using TLM, a component may communicate via its interface to any other component that implements that interface. Thus, one component may be connected at the transaction level to another implemented at multiple levels of abstraction. The common semantics of TLM communication permits components to be swapped in and out without affecting the rest of the environment. The OVM has built-in checking to ensure that communication paths are defined and connected correctly.

Phasing and Execution Management

To be able to have VIP from multiple sources work together in a single environment, the OVM defines a series of phases that all verification components go through during the execution of a simulation (see Figure 4). Once the top-level environment is constructed in the *new()* phase, child components are then instantiated, constructed, and configured hierarchically during *post_new()*. The connections between components are defined during the *elaboration()* phase, and these connections are checked and resolved in the post_elaboration() phase. At the end of *post_elaboration()*, all components in the environment are allocated, connected, and ready for use.

Next, the *pre_run()* phase allows the test to further customize and configure verification components and/or the design under

Figure 4: Simulation Phases of the OVM

test (DUT) prior to executing the test in the *run()* phase. At the conclusion of *run()*, which is a task, three reporting phases are executed: *extract()* allows results to be gathered from specific components, *check()* validates the results to determine the pass/fail status of the test, and *report()* lets each component report its results and status to a log file or the display, using the message severity and formatting routines.

It is possible to define user-, project-, or company-specific phases and insert these into the default list. This ensures they are run on every component at the appropriate time during the simulation. The execution of phases is managed automatically throughout the environment at every level of the hierarchy.

Testbench/Environment Customization

The OVM does not require instantiation of individual components, as in the following example.

```
red my_a;
my_a = new(“a”,this); // hard-coded instantiation/al-
location
```

Instead, the OVM facilitates reuse and customization through a *class factory*, which is a standard object-oriented technique allowing components to be instantiated and initialized on-the-fly from a central location as follows.

```
ovm_component cmp;
cmp = create_component(“red”, “a”); // flexible instan-
tiation
$cast(my_a,cmp);
```

All components in the OVM are extended from *ovm_component*, which is the return type of the *create_component()* method of the class factory. Thus, the return value must be cast to the specific component (*my_a*) that, by default, is now of type *a_c*, as it was in the original code. The class factory also allocates and initializes the component before returning it. The advantage of using the class factory is that it may be overridden to customize the environment.

```
ovm_factory::set_type_override(“red”, “blue”);
ovm_factory::set_inst_override(“top.a”, “red”, “green”);
```

The *set_type_override()* method tells the class factory to return a component of type *blue* whenever a *red* is requested. The environment coded in this example now has a component of type *blue* for *my_a* (and any other instance of the *red* class), allowing a different set of behaviors without changing the environment code, simply because the environment was written to allow this flexibility. The use of TLM interfaces between components facilitates this capability by enforcing the encapsulation of communication. As long as *blue* has the same interfaces as *red*, the rest of the environment is perfectly compatible. Similarly, the type returned by the class factory may also be overridden on a per-instance bases using *set_inst_override()*.

Configuration

One of the keys to reuse is being able to customize and configure components based on their context. The OVM manages this process by allowing components to specify configuration information for their children. During the execution of the *post_new()* phase, each component is responsible for checking whether its internal properties have been so configured, and, if they have, it sets those properties to the configured values. The build process continues top-down, allowing each component to modify the configuration and/or instantiation of its child components.

```
class my_env extends ovm_env;
  block b;
 ...
  function void build(); // called from post_new()
    set_config_int(“b”, “is_active”, 0);
    ...
    b.build();
  endfunction
endclass

class block extends ovm_component;
  driver d;
  bit is_active = 1;
```

```
  function void build();
    if(get_config_int("is_active", is_active)
      if(is_active) begin
        cmp = create_component("driver", "d");
        $cast(d,cmp);
        d.build();
      end
  endfunction
endclass
```

In this example, the environment directs the configuration for the block to set its *is_active* bit to *0*. All OVM components are responsible for getting their own configuration information, so in its build process, the block component checks to get the value of *is_active* from the global table. If a value is found, it is used; otherwise, the default value is used. Configuration values may be set for *int*, *string*, and *ovm_object* parameters, so this simple text-based interface (including wildcarding of names) can be used to set the configuration value for any information required, depending on the component being used.

In this case, the block is set to be inactive (*is_active == 0*). The block component continues its build process by using this configuration information to control the configuration and set up of its children. In this case, it uses the value of *is_active* to control whether to instantiate a driver component. This single block component may now be reused and controlled in many different environments, each of which may choose to activate the driver or not. By designing components to provide such flexibility, it becomes straightforward to create (or purchase) a library of verification components that may be reused without altering its internal contents.

Sequential Layered Stimulus

A third way that the OVM facilitates the customization of particular tests is through the specification of the actual stimuli that will be executed. The OVM enables the rapid creation of interesting transaction stimulus patterns without requiring detailed knowledge of the verification environment infrastructure. This important feature provides allows non-verification experts to quickly create interesting test scenarios that can be reused across multiple tests, verification environment topologies, and projects. Sequential stimulus can range from a purely directed approach to a constrained-random approach that allows constraint layering via class factories, such as that described above.

Once defined, stimulus sequences can be reused as a subset of other stimulus sequences in order to create larger and more interesting test scenarios. Various scenarios can be executed in order to exercise the design with interesting mixes of the stimulus sequences. Complex protocols can be modeled by layering sequences in a hierarchical fashion that provides clean abstraction for each level of the hierarchy. For large verification environments, multiple interfaces can be controlled and coordinated from a central mechanism; known as a virtual sequence.

Summary

Successful verification projects require more than a standard language. A sophisticated methodology is needed to build leading-edge testbenches, ensure interoperability, and promote verification reuse. With several widely used but incompatible verification methodologies available, the industry has been clear in its desire for cooperation among EDA vendors to end the "methodology wars."

The co-development and endorsement by Mentor and Cadence give the OVM credibility and viability as the answer to the industry's concerns. The OVM is clearly the only interoperable, open, and proven verification methodology. With the release of the OVM, there is no longer a methodology war raging. The OVM is already the clear winner.

cadence™
2655 Seely Avenue
San Jose, CA 95134
Phone: 408.943.1234
Fax: 408.428.5001
www.cadence.com

Mentor Graphics®
8005 SW Boeckman Road
Wilsonville, OR 97070-7777
Phone: 503.685.7000
Fax: 503.685.1204
Sales and Product Information
Phone: 800.547.3000
www.mentor.com