

# VMM application packages: the next level of productivity

A STANDARDIZED VERIFICATION METHODOLOGY INCREASES OUTPUT WITHOUT SACRIFICING DESIGN QUALITY.

The history of the EDA industry shows a clear, repetitive pattern. Designers develop new proprietary technologies, leveraging de facto and sanctioned industry standards; leading-edge users identify the most effective of these new technologies; and the industry turns the knowledge that users gain into the next set of de facto or sanctioned standards, allowing the creation of a set of newer technologies.

From the first netlist languages to Verilog to SystemVerilog, the functional verification of the digital-design segment of the EDA industry has seen—and continues to see—a rich sequence of ever-more-powerful standards. The methodology and associated support classes that the VMM (*Verification Methodology Manual*) for SystemVerilog describes were the next logical steps in this constant industrial evolution (Reference 1). Arising from proprietary methodologies that Synopsys ([www.synopsys.com](http://www.synopsys.com)) and ARM ([www.arm.com](http://www.arm.com)) developed, VMM has become a de facto standard for implementing constrained random-verification environments in SystemVerilog.

A standard verification methodology enables the industry evolution to continue to the next level of productivity. Just as VMM's creators built it on SystemVerilog, you can now build application packages on the infrastructure that the VMM provides (Figure 1). User testbenches can then leverage the functions that those application packages provide, making them easier to implement correctly and increasing efficiency.

## IDENTIFYING APPLICATION PACKAGES

Designers build application packages, such as verification environments, on the generic VMM infrastructure. But unlike verification environments, which are specific to a design under verification, application packages are useful in implementing a variety of verification environments. Application packages must thus correspond to needs and requirements for verifying designs. Many years of verifying designs using the VMM and implementing VMM-compliant verification intellectual property—and many customers who do the same—

helped Synopsys identify application packages that can boost the productivity of implementing VMM-based verification environments.

Almost all verification environments share some requirements. For some of these requirements, a simple extension of the VMM standard library is sufficient. For example, every verification environment must determine when it is proper to terminate a simulation. To that effect, VMM creators added a new utility class, `vmm_consensus`, to the VMM standard library. Some of these requirements require additional methodology statements, which new utility or base classes support.

For example, the new VMM environment-composition package lets you reuse block-level environments in system-level environments. Another example is the RAL (register-abstraction-layer) package, which provides an object-oriented mechanism for accessing fields, registers, and memories regardless of their physical locations.

Verification environments share some requirements and use them to verify designs in application domains. An application domain with a large number of designs creates the opportunity for additional application packages. For example, a data-stream-scoreboarding package greatly facilitates the implementation of self-checking structures for networking and DSP applications. Another example is the memory-allocation-management package, which facilitates the allocation of memory buffers for configuring and verifying DMA and memory-controller applications.

ample, a data-stream-scoreboarding package greatly facilitates the implementation of self-checking structures for networking and DSP applications. Another example is the memory-allocation-management package, which facilitates the allocation of memory buffers for configuring and verifying DMA and memory-controller applications.

## END OF TEST

The `vmm_env::wait_for_end()` method aims to let users implement how a verification environment detects the end of a test. Once the task completes, the user can cleanly shut down the verification environment, and a final accounting of all stimuli ensures that the environment has not accidentally lost any stimuli. You can detect the end of test by determining an absolute amount of elapsed time, counting a number of clock cycles, or observing a minimum number of transactions on an interface. These imperative end-of-test conditions are

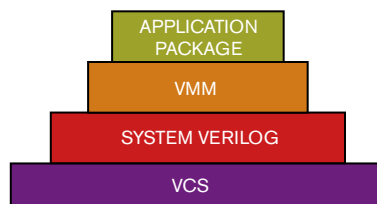


Figure 1 Just as VMM's (*Verification Methodology Manual's*) creators built it on SystemVerilog, you can now build application packages on the infrastructure that the VMM provides.

easy to implement and suitable for directed test cases, but they are usually too simplistic for constrained random-verification environments.

When creating a constrainable random-verification environment, it is difficult to predict exactly how long various tests must run. Some trivial tests may need to run for only a few transactions. Corner-case tests may need to run for several thousand transactions. Furthermore, in a layered-verification environment, it is usually insufficient to count the number of occurrences of a significant event at a single location. For example, the number of bus packets you inject into the environment or the number of bus cycles you observe on an output interface will provide information about only the activity at that location. To safely end a test case in a layered-verification environment, it is usually necessary to wait for a combination of conditions: all of the generators generating the number of required transactions, all transactors sitting idle, no transactions remaining in transaction-level interfaces, or the scoreboard's observing enough activity, for example.

The VMM standard library provides a new utility class, `vmm_consensus`, to facilitate the identification of the test's end. As the name implies, this utility class implements a centralized decision-making mechanism that indicates when no participants object to the decision to end the test. This mechanism is perfectly scalable, allowing verification environments to grow—or for you to combine them—without affecting the complexity of the end-of-test decision. Yet, it transparently makes sure that the test ends as soon as but only when appropriate. For convenience, the `vmm_env` class now includes an instance of the `vmm_consensus` in the `vmm_env::end_vote` property.

The `vmm_consensus` utility class handles a variety of participants (Figure 2). Each participant can then object or consent to the final decision, independently of all other participants. The application can register channel instances as participants that implicitly consent when they are empty. It can register transactor instances as participants that implicitly consent while they are indicating the `vmm_xactor::xactor_idle` notification. The application can register on/off notifications as participants that implicitly consent while on or off. It can register other `vmm_consensus` instances as participants that implicitly consent when all of the other participants consent. You can also obtain generic participant interfaces for user-defined agreement or objection to the decision.

You need no longer implement a complex decision-mak-

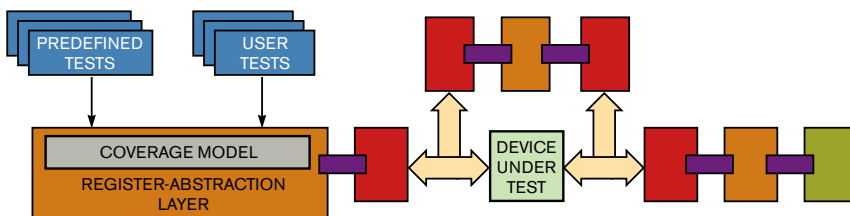


Figure 3 The register-abstraction layer isolates the upper layers of a verification environment and the test cases running on it from the implementation details of accessing the registers and memories in the design.

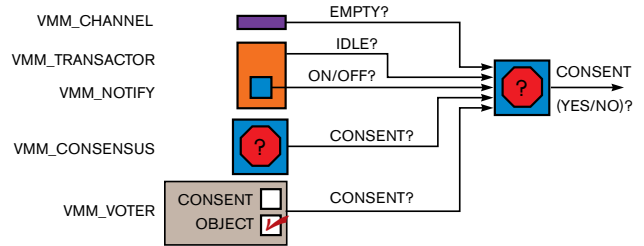


Figure 2 The `vmm_consensus` utility class handles a variety of participants.

ing algorithm with multiple forked threads watching for different end-of-test conditions. Furthermore, because an object encapsulates the decision making, an environment can pass it to subenvironments, so the end-of-test decision process can scale as the complexity of the system-level-verification environment increases.

Because the class is a generic decision-making utility, an environment can use different instances of the `vmm_consensus` to implement other decisions. For example, a transaction-level model could use the class to determine when all concurrent processing threads are done and thus can accept a new transaction for processing. A distributed simulation-sequencing mechanism could use it where various components indicate their objection or consent to proceeding further in the simulation.

## REGISTER-ABSTRACTION LAYER

Almost every design possesses registers accessible through a host processor. The first test cases written to verify almost every design are to determine that those registers operate as you would expect: that they reset to the appropriate value, that you can write writable bits, and that you cannot modify read-only bits. Unique to each design are the number of registers, the fields that compose them, whether you can write or read fields, and the physical protocol you use to write or read them.

When a design contains a few dozen registers, you can manually create—and, more important, maintain—the test cases and firmware-emulation routines. Using symbolic values for addresses and bit offsets can compensate for the maintenance effort. But when the number of registers is in the hundreds or thousands, creating and maintaining the register-correctness test cases and the firmware-emulation routines that the other functional tests use become overwhelming.

The task gets even more daunting when it becomes necessary to migrate the block-level environment, firmware emulation, or tests to a system-level environment: The address of every register differs, and the physical interface you use to access the registers may no longer be available.

At this point, most verification teams determine that automatically generating some abstracted mecha-

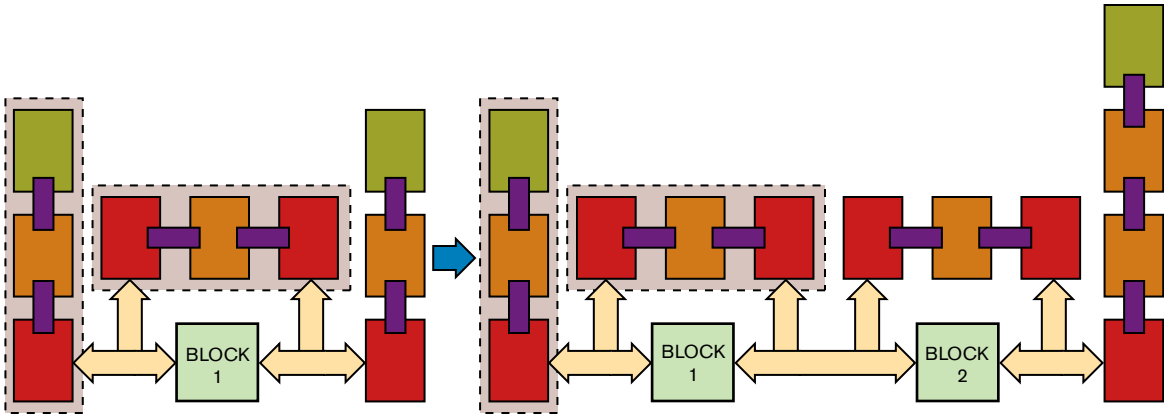


Figure 4 You can reuse entire networks of transactors from one environment to another.

nism for accessing and verifying the correct operation of the registers from a specification is more efficient than manually implementing and maintaining them. A generic register-access-abstraction mechanism, along with an automatic generation process from an executable specification of the registers in a design, dramatically reduces the threshold at which this automation becomes more productive.

The VMM RAL application package is a set of base classes, a code generator, and an executable-register-specification format that enables the automated generation of an object-oriented abstract register-access mechanism and predefined register-operational tests. The RAL isolates the upper layers of a verification environment and the test cases running on it from the implementation details of accessing the registers and memories in the design (Figure 3).

RAL simplifies not only the maintenance and verification of accessing registers and memories in the design, but also writing the code that needs to access them. Changes in physical interface, address, location, or bit offset of a field do not affect this simpler code, because the automatic generation of the design-specific RAL model based on any modification to the executable specification takes care of these nonfunctional implementation details.

The executable specification for the register that the RAL application package uses is congenial enough that you may use it as the primary register-specification vehicle and manually author and maintain it. However, you can just as easily gen-

erate it from register-specification documents, such as Excel spreadsheets or Word documents.

## ENVIRONMENT COMPOSITION

The VMM defines the transactor as the unit of reusability, but more complex structures can be reusable. For example, a complete TCP/IP (Transmission Control Protocol/Internet Protocol) stack can be reusable, or the interface monitors and self-checking structure of an intellectual-property core can also be reusable. You can reuse entire networks of transactors from one environment to another (Figure 4).

Why not simply reuse entire environments? Why bother with subenvironments at all? Environments are specific to a design under verification. Unless a design is identical to the design you previously verified, the verification environment is not reusable. You must introduce configurability into the original environment to support its reusability in different contexts. For example, you would have to disable the entire stimulus stack on the SOC (system-on-chip)-bus side of an intellectual-property core to reuse the environment from the core level to the SOC level.

Rather than promote the creation of highly configurable verification environments, this application package takes a middle-of-the-road approach. Environments are design-specific and, thus, not reusable as is. However, you may make portions of those environments reusable and configurable. This ability simplifies the correct construction of both the reusable subenvironments and the environments that use them. But, should a complete environment be reusable, nothing prevents an author from encapsulating it as a reusable subenvironment.

Furthermore, subenvironments are not limited to encapsulating transactors, channels, and physical interfaces. They can also encapsulate smaller subenvironments that you combine to create a subsystem environment. You can then reuse these subenvironments in system-level environments.

Creating reusable subenvironments entails more than simply wrapping transactors, channels, and physical interfaces in yet another class. It must be possible to configure the DUT (device-under-test) functions associated with the reused subenvironment within the new context, without modifying the block-level configuration code. The VMM

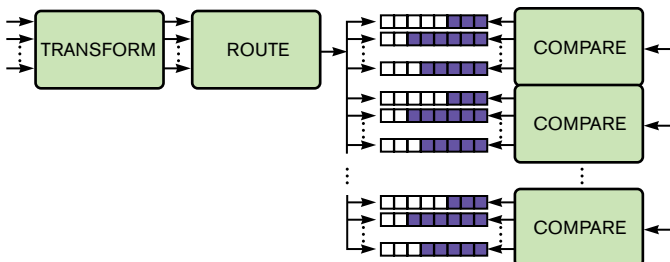


Figure 5 You can also use the VMM data scoreboard to verify multistream designs with user-defined data transformation and input-to-output stream routing.

RAL abstracts the configuration process in the subenvironment from the physical details of writing registers and memories. If the block-level functions require access to memory that the system may share at the system level, a memory-allocation manager must ensure that different blocks use different memory regions. Identifying the end-of-test condition must take into account the needs of the various subenvironments without requiring the author of the system-level environment to know the details of how a subenvironment determines it is time to end the simulation. The new `vmm_consensus` utility class allows the end-of-test condition to scale from block- to system-level environments.

Reusing various structures from block- and system-level verification environments does not happen by accident. It is necessary to plan and build those block-level environments to create reusable structures. The VMM environment-composition package provides clear guidelines and support for implementing these reusable structures. Reusable subenvironments can greatly accelerate the creation and maintenance of higher level environments. With true reuse, you can even execute the development of block- and system-level environments in parallel, with the system-level environment automatically inherit-

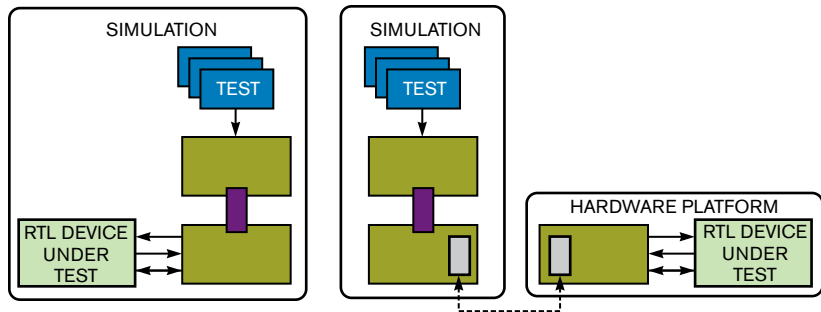


Figure 6 You can easily target a layered VMM-based verification environment from a pure software-simulation engine to a coemulation engine.

ing all updates and improvements you make to the portions of the block-level environments it reuses.

### DATA-STREAM SCOREBOARDING

Implementing the response-checking mechanism in a self-checking environment remains the most time-consuming task. The VMM data-stream-scoreboarding package facilitates the implementation of verifying the correct transformation, destination, and ordering of data streams. This package is intuitively applicable to packet-oriented design, such as modems, routers, and protocol interfaces. You can also use it to verify any design transforming and moving sequences of da-

ta items, such as DSP datapaths and floating-point units.

You can use the VMM data-stream scoreboard out of the box to verify single-stream designs that do not modify the data flowing through them. For example, you can use it to verify FIFOs, MACs (media-access controllers), and bridges.

You can also use the VMM data-stream scoreboard to verify multistream designs with user-defined data transformation and input-to-output stream routing (Figure 5). For example, it can verify an encryption engine as single stream or as a crypto transformation; as routers for multistream routing tables or with no transformation; and as data multiplexers with multiple input streams or encapsulation. The transformation from input

data items into expected data items is not limited to one-to-one transformations. You can transform an input data item into multiple expected data items, such as segmenters, or no expected data items, such as reassemblers.

The package provides three predefined “expect” functions: strictly in order, in order with losses, and out of order. But, should you require a different expect function, powerful iterator classes allow you to easily traverse the scoreboard-data structure without knowing the details of its implementation. You can also use these same powerful iterators to implement more complex prediction functions, such as reordering or multicasting.

Using the VMM data-stream scoreboard simplifies the creation of a self-checking structure by leveraging a proven and efficient set of data-structure and look-up functions. It also facilitates integrating the scoreboard with the rest of the verification environment. The VMM data-stream scoreboard is aware of the methodology for implementing the verification envi-

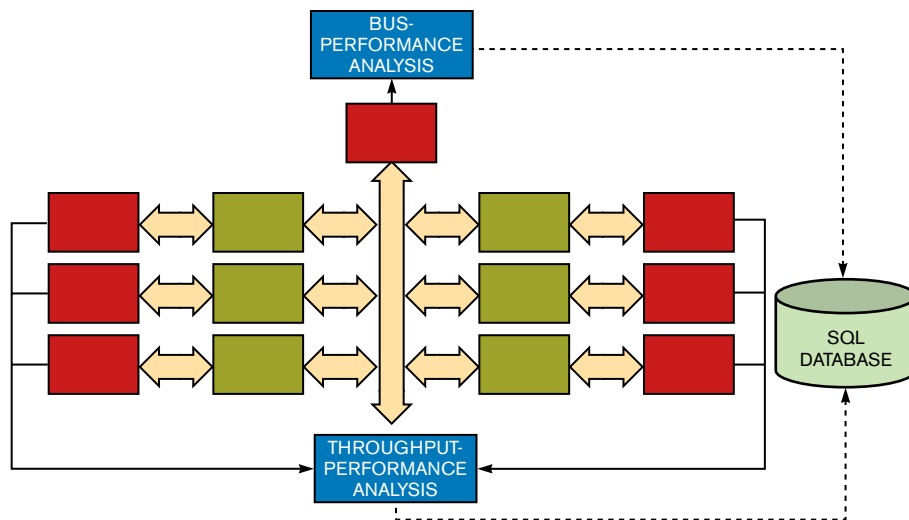


Figure 7 The VMM performance-analysis package can measure the performance of an SOC bus with multiple masters and multiple slaves or measure the overall throughput of a design.

ronment in which it will see use, and you can make VMM-compliant transactors aware of the scoreboard that will verify the response. The latest Synopsys version of the VMM library therefore includes predefined integration methods and macros to ease the task of getting the stimulus and observed response transactions to the scoreboard. Instead of manually extending, instantiating, and registering callback methods, VMM-compliant transactors now offer a predefined scoreboard-integration method. For example, you can now integrate the predefined VMM generators with a VMM data-stream-scoreboard instance using a single method call. Similarly, you can easily attach a VMM data-stream-scoreboard instance to the status information of a VMM notification.

## HARDWARE-ASSISTANCE LAYER

Simulations that require days of runtime can benefit from the performance of hardware-assisted verification. You map

the design to a reconfigurable hardware platform, and the testbench remains on the simulator. The simulator and hardware platform then communicate to execute the simulation. The system realizes the greatest performance benefits when you also map a portion of the testbench to the hardware platform, alongside the design, and you minimize the frequency and amount of communication between the hardware platform and the simulator.

You can easily target a layered VMM-based verification environment from a pure software-simulation engine to a coemulation engine (**Figure 6**). The SCE-MI (Standard Coemulation Modeling Interface) industry standard offers a standard transaction-level communication mechanism between the simulated and the emulated layers of the testbench. However, the SCE-MI standard is currently only for C testbenches. The ability to reuse the SystemVerilog testbenches to verify the basic functions of the design and its components provides for greater productivity and shorter time to emulation.

The VMM hardware-abstraction-layer package provides a SCE-MI-like transaction-level interface mechanism between a simulated SystemVerilog testbench and emulated synthesizable bus-functional models, also written using SystemVerilog. The advantage of using a common language for both sides of the coemulation equation is that you can create a reference implementation that does not use a hardware platform. You can then use the reference implementation to develop the testbench and test cases as well as to reproduce failures that you

observed on the emulator. This reference implementation, although significantly slower than a hardware platform, provides for a more flexible and integrated debugging capability, without tying up the hardware platform.

The VMM hardware-abstraction-layer package includes several guidelines for building coemulation testbenches and writing synthesizable command-layer transactors. It provides synthesizable message interfaces for synthesizable transactors and message-port classes to send messages to and receive messages from the emulated transactors. These interfaces and classes encapsulate the platform-specific message-passing mechanism, which can be SCE-MI, and render the verification environment portable to hardware-assistance platforms.

## PERFORMANCE ANALYZER

Functional coverage, which encompasses functional-coverage points in cover-group statements or coverage properties, measures the occurrence of simple singular events. When an event happens, the system records it. The simple fact that an event has occurred is enough to satisfy functional-coverage points.

But singular events are just one type of functional-coverage metric. The functional correctness of many designs requires the collection of statistical-coverage metrics, such as minimum, maximum, and average memory-request latencies or average and peak bus usage.

The VMM performance-analysis package enables the col-

lection of statistical-coverage metrics. It can measure the performance of any resource that you use over time. Multiple initiators can use the resource, and multiple targets may provide the resource. For example, you could use it to measure the performance of an SOC bus with multiple masters and multiple slaves. You could also use it to measure the usage of a shared memory or an arbiter of the overall throughput of a design (Figure 7).

A simulation can contain several performance-analyzer instances, each measuring different performance aspects. You can merge the performance metrics you collect during concurrent simulation runs of a regression suite to obtain overall performance measurements across all of the simulations.

The definition of useful work and when that work starts, stops, resumes, and eventually completes are up to the user, as is the measure of usefulness of that work. Users can measure performance with respect to simulation time, number of clock cycles, or any other gauge of duration the user chooses. You can thus use the VMM performance-analysis package to measure any activity that occurs, such as bus usage, memory-subsystem response, or interrupt servicing, over time. The performance analyzer keeps track of the metrics—whatever they signify to the user—and allows the system to record, merge, and report them.

The system verifies the functional correctness of a design using scoreboarding techniques or reference models. When exploring the architecture of a design, functional correctness also includes meeting the performance requirements of the design. You can use the performance-analysis package to determine the performance correctness of a chosen architecture on a transaction-level or stochastic model of the design architecture or on an RTL (register-transfer-level) model.

Standardizing a common verification methodology offers the opportunity for developing functions at a higher level of abstraction. The VMM application package allows a verification team to increase its productivity without sacrificing the quality of the design under verification. **EDN**

---

## REFERENCE

■ Bergeron, Janick, Eduard Cerny, Alan Hunter, and Andy Nightingale, *Verification Methodology Manual for SystemVerilog*, Springer, 2005, ISBN 978-0-387-25538-5.



## AUTHOR'S BIOGRAPHY

Janick Bergeron helps define the state of the art in functional-verification methodology and the tools that support it. He is the author of *Writing Testbenches Using SystemVerilog*, co-author of *Verification Methodology Manual for SystemVerilog*, and the moderator of the *Verification Guild*. Before joining Synopsys, Bergeron was chief technology officer of Qualis Inc and a member of the scientific staff at Nortel Networks. He holds a master's degree in business administration from the University of Oregon (Eugene, OR), a master's degree in electrical engineering from the University of Waterloo (Ontario, Canada), and a bachelor's degree in engineering from the Université du Québec à Chicoutimi (Quebec, Canada). He has been with Synopsys since 2003.