cadence®

# Comprehensive UVM/OVM Acceleration

Universal Verification Methodology (UVM)/Open Verification Methodology (OVM) acceleration offers a consistent, easy-to-follow use model for accelerated and simulated environments. It extends the UVM/OVM to enable acceleration-ready universal verification components (UVCs) and a metric-driven verification (MDV) flow. UVM/OVM acceleration places an emphasis on reuse and automation.

## Contents



## Introduction

Today's traditional design flow involves design at multiple levels of abstraction. As the design implementation is refined and the verification vehicle changes, the testbench needs to adjust to abstractions from transaction-level simulation, and RTL simulation to hardware acceleration. But the challenge is that separate verification components, testbenches, tests, and plans are developed by separate teams at different abstraction levels (Figure 1). This requires more expertise to learn and understand, more code to develop and maintain, and larger teams to fund and manage.

This white paper describes a consistent and comprehensive Universal Verification Methodology (UVM)/Open Verification Methodology (OVM) that prescribes substantial reuse across multiple levels of abstraction and facilitates a metric-driven verification (MDV) flow. While this document focuses on UVM/OVM acceleration, it also touches on other abstraction levels, such as transaction-level modeling (TLM).

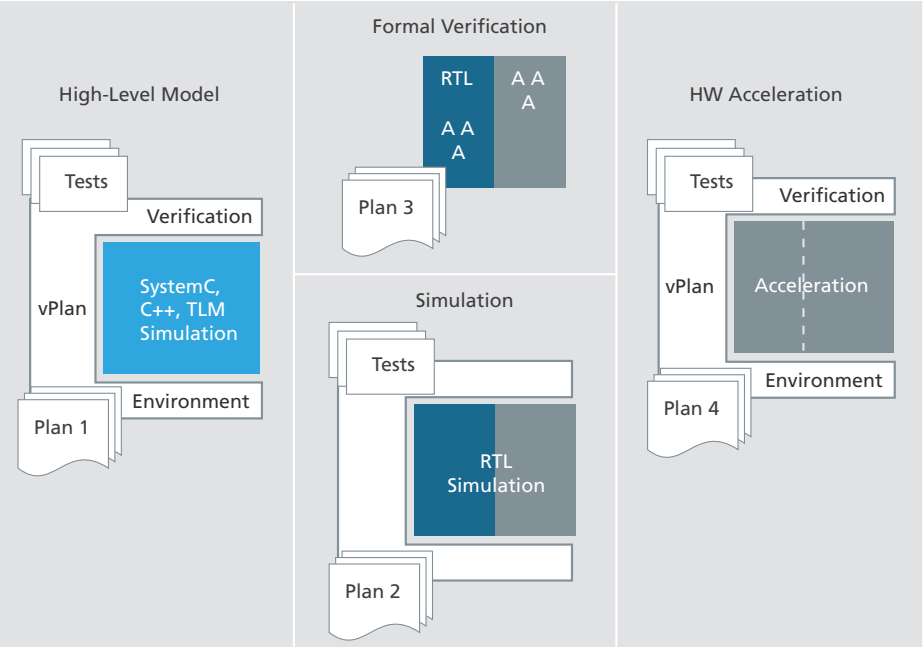


*Figure 1: Typical development flow today*

## The UVM/OVM for Simulation and Acceleration

With the ever-growing design size and complexity of modern automated testbenches used for coverage-driven verification, the need for hardware acceleration grows as well as the need to shorten the verification schedule. An ideal solution makes use of an advanced, automated verification environment with constrained-random stimulus, functional coverage, and separate scoreboard checking, and reaps the benefits of acceleration using a hardware platform. Two complementary approaches to handling design complexity are modern automated SystemVerilog or ***e*** coverage-driven testbenches for functional coverage with intelligent stimuli creation, and hardware acceleration for pure speed. Engineers need a solution that leverages the benefits of both approaches.

The traditional challenge in simulation acceleration is to partition the environment such that frequently active register transfer level (RTL) logic is placed in the hardware accelerator, while high-level computation logic that cannot be accelerated is placed in the software simulator. The amount of logic placed in the software simulator should be kept to a minimum in order to limit the time spent simulating it. Engineers must also minimize the frequency of communication context switches between the accelerated and simulated parts. An important requirement is to devise a solution that leverages the UVM/OVM methodology and use model. Adherence to UVM/OVM recommendations for packaging, instantiation, configuration, and test writing reduces the knowledge needed for running such testbenches, facilitates more code reuse, decreases the usability gaps between the acceleration and simulation teams, and enables a consistent company-wide methodology. Using the UVM/OVM is also an opportunity to drive the industry to produce more acceleration-ready verification components (UVCs), which can be used for simulation as well as hardware acceleration.

Note: All references to UVCs in this document are used in conjunction with the UVM. With the OVM, these verification components are known as OVCs.

### UVM/OVM acceleration deliverables

UVM/OVM acceleration enables a metric-driven verification (MDV) flow for accelerated environments and includes a methodology guide, class library enhancements (for both OVM-SystemVerilog and OVM-e), a reference manual, and complete examples on how to accelerate UVM/OVM environments. The class library is built on standards, both transaction-level modeling (TLM) and the Accellera Standard Co-Emulation Modeling Interface (SCE-MI) version 2.0. It is consistent with pure software simulation UVM/OVM and provides automation (for example, built-in serialization capabilities and dynamic controllable message buffering to trade off controllability with speed). UVM/OVM acceleration is language-independent and is supported in both the ***e*** and SystemVerilog languages. The hardware transactors (or accelerated bus functional models)—which decompose untimed high-level messages from simulation to a series of cycle-accurate signal-based events on the acceleration platform or, conversely, compose a series of signal-based events from the acceleration platform into a single high-level message—can be used by both languages. Cadence® acceleration technology is designed and tuned to enable UVM/OVM acceleration requirements for development, simulation, and debug.

## Creating Reusable Verification Components

The UVM/OVM encourages companies to build a central repository of reusable verification components (UVCs) for protocols (interface UVCs) and for subsystems (module UVCs). Over time, this repository becomes bug-free—optimized and reliable. The goal of the comprehensive UVM/OVM is to devise a multi-purpose UVC architecture that supports designs in various abstraction levels and operation modes. Acceleration-ready UVCs are generic, configurable verification environments that are proven to work on both simulation and acceleration platforms. The next section covers the guidelines for implementing acceleration-ready verification components.

### Partitioning the agent's simulated and accelerated logic

Obviously, the accelerated logic resides within the accelerator, is executed at hardware speed, and is stimulated by a transaction-level testbench via SCE-MI. The question remains about how to create UVCs that can work on both software and hardware environments while maintaining the same instantiation and use model. On the injection path, the UVM/OVM already differentiates between transaction-level and signal-level activity using sequencer and driver components.

But in today's verification environments, collection and monitoring reside within a single monitor component. UVM/OVM acceleration recommends a split monitor architecture (Figure 2) in which a signal-level collector gathers the signal activity and forms transactions. The monitor activity remains the same for performing protocol transaction-level checking and coverage. The advantages of this architecture include separation between transaction-level checking/coverage, signal-level activities, and a clean topology to move among the different levels of abstraction needed for both TLM and acceleration.

An enumerated type called ovm_abstraction_level_enum provides the configuration switch to move among the different levels of abstraction. This enum is similar to the active/passive knob and is set via the OVM configuration mechanism to use the accelerated or simulated logic.

One decision users need to make is whether they are going to implement a single synthesizable and acceleratable bus functional model (BFM) or separate BFMs for simulation and acceleration. Implementing a single BFM typically reduces the need to maintain two versions. However, synthesizable BFMs are more time-consuming and difficult to create and debug. Many times, for acceleration and system-level purposes, users create a much simpler BFM that does not deeply exercise the protocol correctness. Hardware-implemented BFMs cannot be extended (which can result in cut-and-paste reuse with even more code to maintain); using them in a simulation environment changes the environment topology and sometimes the simulated driver already exists. Many users choose the dual BFM approach for the aforementioned reasons, but users should decide on a case-by-case basis whether to use a single BFM or dual BFMs.

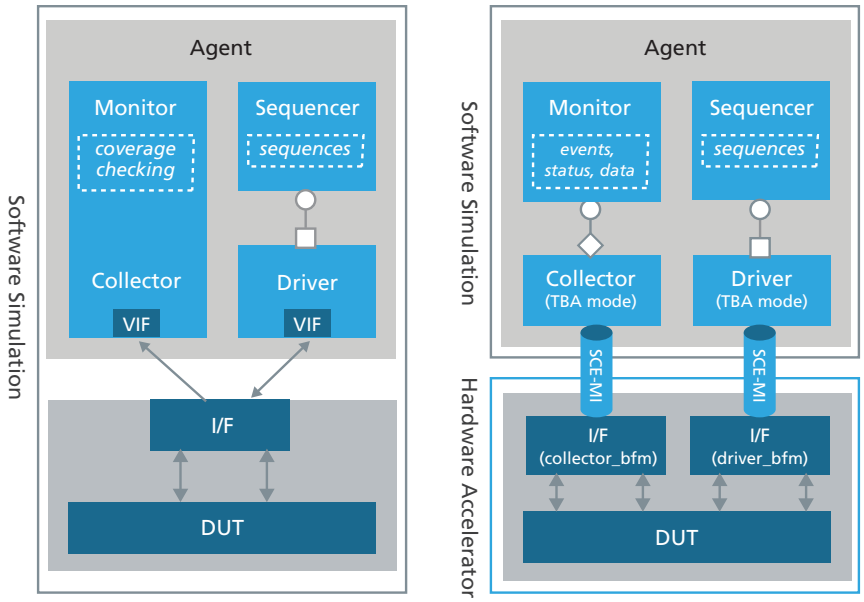


*Figure 2: A split monitor architecture*

```
// File: Accel_lib/yamp/sv/yamp_master agent:sv
class yamp_master_agent extends ovm_agent;
   protected ovm_active_passive_enum is_active = OVM_ACTIVE;
   ovm_abstraction_level_enum abstraction_level = OVM_SIGNAL;
   yamp_master_monitor  monitor;
   yamp_master_collector  collector;
   yamp_master_sequencer  sequencer;
   yamp_master_driver  driver;

function void build();
   super.build();
   set_config_int("driver", "abstraction_level",
abstraction_level);
   set_config_int("monitor", "abstraction_level",
abstraction_level);
   monitor =
yamp_master_monitor::type_id::create("monitor", this);
   if (is_active == OVM_ACTIVE) begin
      sequencer =
yamp_master_sequencer::type_id::create("sequencer", this);
   driver = yamp_master_driver::type_id::create("driver", this);
   end
endfunction : build
```

## Hardware/software buffered communication using SCE-MI proxies

The SCE-MI 2.0 standard provides an efficient, vendor-independent solution to communicate with hardware acceleration platforms. Using SCE-MI directly in UVM/OVM simulation is foreign to UVM/OVM users since it imposes its own API and is not runtime-configurable. UVM/OVM simulation users are more familiar with concepts like TLM and runtime configuration. UVM/OVM acceleration extensions provided by Cadence provide SCE-MI pipe proxy base-class components to communicate transactions between hardware and software. On the UVM/OVM side, the proxies provide a native ***e*** or SystemVerilog TLM interface, which abstracts away the SCE-MI interface. On the other end, the proxies connect to the SCE-MI in the accelerated testbench modules collector_bfm and driver_bfm (Figure 3).

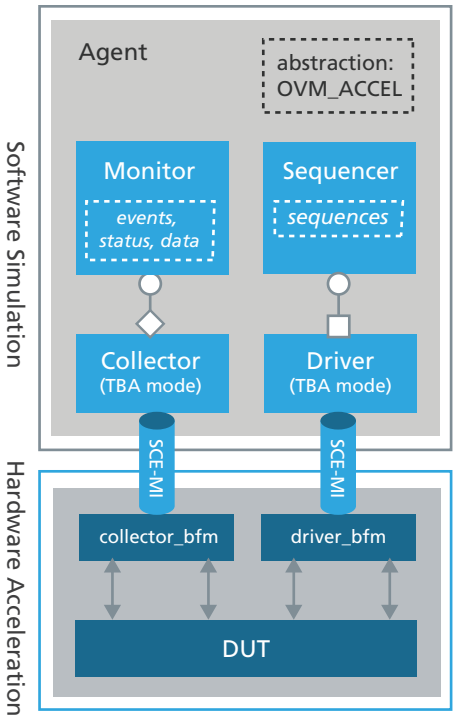


*Figure 3: SCE-MI pipe proxy setup*

OVM acceleration proxies also provide automatic serialization and buffering of objects using OVM-packed fields. A few of the key proxy requirements were to avoid making users write C code to access SCE-MI 2.0 pipes infrastructure and to be able to efficiently support messages with various sizes. The SCE-MI pipe proxies allow passing dynamic-sized messages and provide a generic C code package that keeps the user in their comfort-zone language. The proxies are components that are parameterized with the data type and a serialization policy class. OVM pack/unpack automation is used by default for serialization and de-serialization, and if needed, users can create their own serialization policies. Proxies are configured via the OVM configuration mechanism.

## Connecting the high-level proxies to the SCE-MI pipes

Binding the SCE-MI pipes in software is done via strings that describe the location of the pipes in the accelerated testbench. Hard-coding the pipe location in the software components breaks fundamental OVM reusability requirements, and also does not scale for multiple component instantiations. For this reason, OVM acceleration demonstrates adding a hdl_path attribute to components that can be set using the OVM configuration mechanism (Figure 4).

```
   / File: Accel_lib/yamp/sv/yamp_master_driver.v
class yamp_master_driver extends ovm_driver #(yamp_transfer);
   // The virtual interface used to drive and view HDL signals.
   virtual interface yamp_if vif;

   // Count transfers sent
   int num_sent;
   protected ovm_abstraction_level_enum abstraction_level = OVM_SIGNAL;
   protected string    m_hdl_path;
   protected ovma_input_pipe_proxy#(yamp_transfer) m_ip; // sending transfer
   protected ovma_output_pipe_proxy#(yamp)transfer) m_op; // receive response
// Additional class methods
   extern virtual task run();
   extern virtual function void build();
   extern virtual protected task get_and_drive();
   extern virtual protected task reset_signals();
   extern virtual protected task drive_transfer(yamp_transfer transfer);
   extern virtual function void report();
   // TBA class methods - notice the tba post_fix
   extern virtual protected task get_and_drive_tba();
endclass : yamp_master_driver
```

```
function void yamp_master_driver::build();
   super.build();
   if (abstraction_level == OVM_ACCEL) // allocation of the scemi pipe proxies
      begin
         set_config_string("m_ip", "hdl_path", {m_hdl_path,".inbox0"});
         set_config_string("m_owp", "hdl_path", {m_hdl_path,".outbox0"})
         m_ip = new("m_ip", this);
         m_op = new("m_op", this);
      end
endfunction : build

// Gets transfers from the sequencer and passes them to the driver.
tasks yamp_master_driver::get_and_drive_tba();
   forever begin
      seq_tiem_port.get_next_item(req);
      // Drive the item
      ovm_report_info(get_type_name(), $psprintf("Driving transfer :\n%s",
req.sprint()), OVM_MEDIUM);
      m_ip.put(req);
      if(req.direction == READ) begin
         m_op.get(req);
         seq_item_port.item_done(req);
      end
      else begin
         // Communicate item done to the sequencer
         seq_item_port.item_done();
      end

      num_sent++;
      ovm_report_info(get_type_name(); $psprintf("Transfer complete :\n%s",
req.sprint()), OVM_MEDIUM);
```

*Figure 4: A master driver code sample*

Using the abstraction-level configuration, the UVM/OVM environment developer can layer the implementation for the accelerated transaction-level driver or collector in the same class (Figure 4). In this way, the existing non-accelerated driver infrastructure is left intact for a possible different use model; e.g., a timing-controlled simulation of the protocol through the virtual interface connection to the device under test (DUT).

## Picking up the pace

The suggested topology and automation described above enables the ultimate goal of accelerating big parts of UVM/OVM testbenches by minimizing the time spent simulating behavioral logic and reducing the communication between the software simulator and the hardware accelerator. Some protocols, such as video or Ethernet packets, lend themselves nicely to acceleration. These include large data items that can be rapidly randomized and have a lot of simulation time to be processed. Various techniques exist to speed up other protocols. These include sending batches of data items, caching possible responses, recording and replaying traffic from files, and more.

## Leveraging existing verification assets

Existing UVCs or synthesizable hardware transactors can be used to significantly speed up the development of acceleration-ready UVCs. To convert an existing UVC to become acceleration-ready, the user needs to:

- Split the monitor into collectors and transaction-level monitors
- Add the SCE-MI 2.0 proxy components
- Add the level of abstraction mode
- Adjust the logic to use newly developed or reused accelerated BFMs

Following a set of basic guidelines on UVC creation can save time later on in re-factoring UVCs for acceleration. For example, having a collector and a monitor is a great practice regardless of acceleration. Users can add the abstraction-level field even if they do not plan to implement all the abstractions and operation modes. In addition, by adopting these guidelines, users can avoid the challenges associated with maintaining two environments—one for simulation and another for acceleration.

## Accelerated UVM/OVM Environments

Conceptually, UVM/OVM testbenches are nothing but the instantiation and configuration of reusable components with some extra glue logic; but, in reality, there is much more than that. Scoreboards, device-specific coverage, register memory, and interrupt handling are all part of a system UVC. Some of the requirements of UVM/OVM acceleration include the ability to leverage exactly the same setup that includes build, hookup, and configuration for accelerated or simulated testbenches. Accelerated testbenches introduce new challenges for instantiation, topology, and build processes. Some of the components that belonged to the dynamic hierarchy need to reside within the static world of modules and interfaces. UVM/OVM examples demonstrate the split of the static and quasi-static worlds, as one module calls the run_test () that eventually bootstraps the dynamic testbench. Another module instantiates the DUT, accelerated BFMs, interfaces, clocks, and resets (Figure 5).

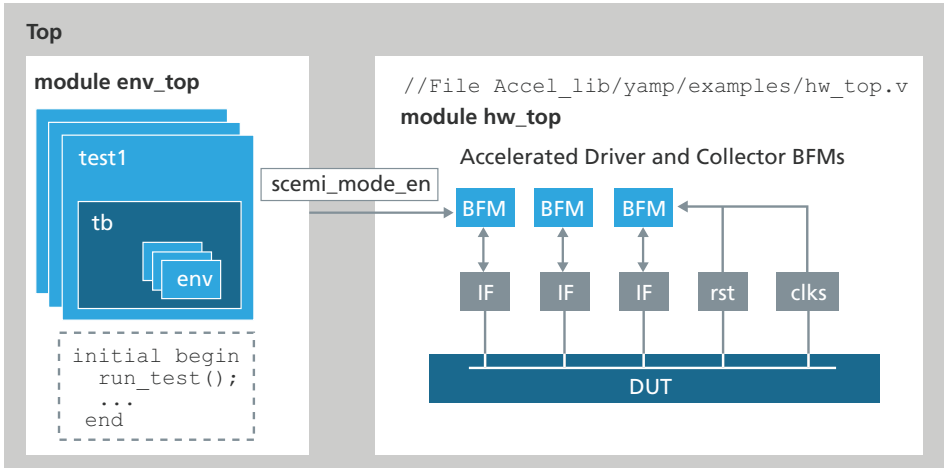


*Figure 5: Testbench env_top and hw_top modules*

This setup of two top modules is useful because it enables the behavioral part of the testbench (which will be simulated) and the RTL part of the testbench (which will be accelerated) to be compiled and optimized separately. This removes the need to re-compile the whole environment when changes are made to components in one of the partitions. Only the affected partition needs to be re-compiled and incrementally elaborated.

### The hardware top module

Separating a hardware top module hw_top is a critical step in UVM/OVM acceleration (Figure 5). The goal of this module's hierarchy: enable high-performance execution of the DUT and verification components (such as BFMs) on the target acceleration platform by allowing the hardware engine to run, which consumes and generates transaction-level data for a UVM/OVM testbench.

The considerations of this hierarchy (Accel_lib/yamp/examples/hw_top.v)are:

1. Signal-based activity: Ideally, there should be no signal-based activity in and out of this hw_top. Such activity should be restricted to some control or configuration signals that do not change later during runtime (e.g., scemi_mode_en signal in Figure 5). This is a performance consideration because frequently changing I/O will significantly reduce runtime performance. The main activity should be at a transaction level.

2. Clocks: Clocks used by the DUT and BFMs should be generated within the hw_top hierarchy. Cadence acceleration engines allow clock specifications in the form of #delay, such as:

```
// File: Accel_lib/yamp/examples/clkgen.v
timescale 1ns/100ps
module clkgen )clk1, clk2);
output reg clk1 = 0;
output reg clk2 = 1;
   always #5 clk1 = ~clk1;
   always #17 clk2 = clk2;
endmodule
```

It is further recommended to keep clock generation in a separate module, as shown above.

3. Reset: Any reset signals and related sequences should be generated within hw_top.
4. Interfaces: As is standard in the UVM/OVM, the DUT is driven using SystemVerilog interfaces. These interfaces also reside in hw_top. They are connected to virtual interface instances in UVM/OVM drivers or collectors when the testbench runs at abstraction_level==OVM_SIGNAL.
5. BFMs: All driver- and collector-acceleratable BFMs should be instantiated in the hw_top partition. They will internally contain one or more instances of SCE-MI 2.0. The BFMs will also use interface instances to interact with the DUT. The BFMs always remain instantiated but are activated only when the testbench environment sets the scemi_mode signal, as shown below. This is done to ensure that they will drive their outputs to Z so as not to conflict with UVM/OVM testbench signal-level drivers in abstraction_level==OVM_SIGNAL.

```
// File: Accel_lib/yamp/sv/yamp_master_driver_bfm.sv
module yamp_master_driver_bfm (clk, len, ce, we, rd, addr,
di, dout, scemi_mode);

   input clk'
   output reg cmd;
   output reg [7:0] len;
   output reg we, ce, rd;
   output reg [15:0] addr, di;
   input [15:0]    dout;
   input scemi_mode;

   scemi_input_pipe #(2, 1) inbox0 ();
   scemi_output_pipe #(2, 1) outbox0 ();

reg cmd_r = 0;
always@ (cmd_r or scemi_mode) cmd <= scemi_mode?cmd_r:1'bz;

reg [7:0] len_r = 0;
always@ (len_r or scemi_mode) len <= scemi_mode?len_r:8'bz;
...

...

endmodule
```

## Controlling the testbench topology

The UVM/OVM configuration mechanism allows randomizing the topology (using a configuration class) or setting it procedurally. The configuration can alter the component hierarchy that is built dynamically at time zero, before the run phase starts. For example, passive agents do not build their sequencers and drivers, or the agent instances number is controlled via a configuration attribute in a multi-drop bus. Accelerated collectors and BFMs hinder this dynamic build process of the topology, as the number of instances is determined in elaboration time using Verilog-generate statements. Topology randomization in that case involves two-step simulation in which the configuration is decided on the first step, and the needed elaboration is achieved on the second step. UVM/OVM acceleration demonstrates a technique that allows a single-phase simulation—a set of instances are pre-elaborated and the simulated testbench determines the required instances by disabling the unnecessary ones. This approach eliminates re-elaboration and allows usage of the UVM/OVM configuration mechanism.

## Cadence UVM/OVM Acceleration-Enabling Technology

Cadence is a leader in hardware acceleration technology, with best-in-class solutions for signal-based acceleration, transaction-based acceleration, and in-circuit emulation. Cadence offers industry-standard (SCE-MI 2.0-compliant) transaction-based acceleration capabilities on all of its hardware-assisted verification platforms, such as the Palladium® series of emulators/accelerators, the Palladium® XP verification computing platform, and Incisive® Xtreme® systems.

Cadence acceleration platforms come with easy-to-adopt features:

1. Co-simulation and dynamic hot-swap with Incisive Enterprise Simulator: This capability allows users to employ transaction-based acceleration of a traditional simulation environment or dynamically hot-swap to and from simulation to high-performance acceleration.

2. Fast turnaround from RTL to acceleration: Using direct compile from RTL to the acceleration engine, users can turnaround a new version of RTL to run onto hardware platforms at the rate of 30-35 million gates per hour using a standard off-the-shelf CPU. Verification teams can keep pace with designers constantly during the RTL design phase, providing timely feedback from accelerated regression runs.

3. Complete visibility and history for debugging: Debug features become important with the unprecedented capability to run millions of cycles per second in transaction-based acceleration mode, driving the design to states unachievable with software simulation alone. Cadence hardware acceleration platforms enable users to debug any part of the design at any point in its design state with 100% visibility from start to finish during an acceleration session.

4. Extending the SCE-MI 2.0 standard to feature-rich HVLs: Not only does Cadence support the C/C++-based SCE-MI 2.0 standard, but it also extends the same capabilities to hardware-verification languages (HVLs) such as object-oriented SystemVerilog and SystemC® languages or aspect-oriented ***e,*** allowing traditional metric-driven verification methodologies to benefit from hardware acceleration. This enables users to run longer tests and debug in a shorter amount of time, thereby improving overall functional coverage before the design is committed to silicon.

5. Extended behavioral modeling subset: For modeling verification tasks to be executed on the acceleration engine, Cadence behavioral compilation technology allows users to write a rich superset of standard synthe-sizable RTL. This reduces the time to develop and maintain acceleratable verification environments while maintaining performance and ease of use.

6. Metric-driven verification: Palladium XP supports an MDV flow, which enables users to follow a comprehensive plan-to-closure methodology that increases verification predictability, productivity, and quality. Verification plans are based on specifications; metrics are constructed with appropriate coverage (UVM/OVM-compliant), checks, assertions, and reuse. Figure 6 depicts the four stages of the metric-driven verification flow (plan, construct, execute, and measure/analyze).

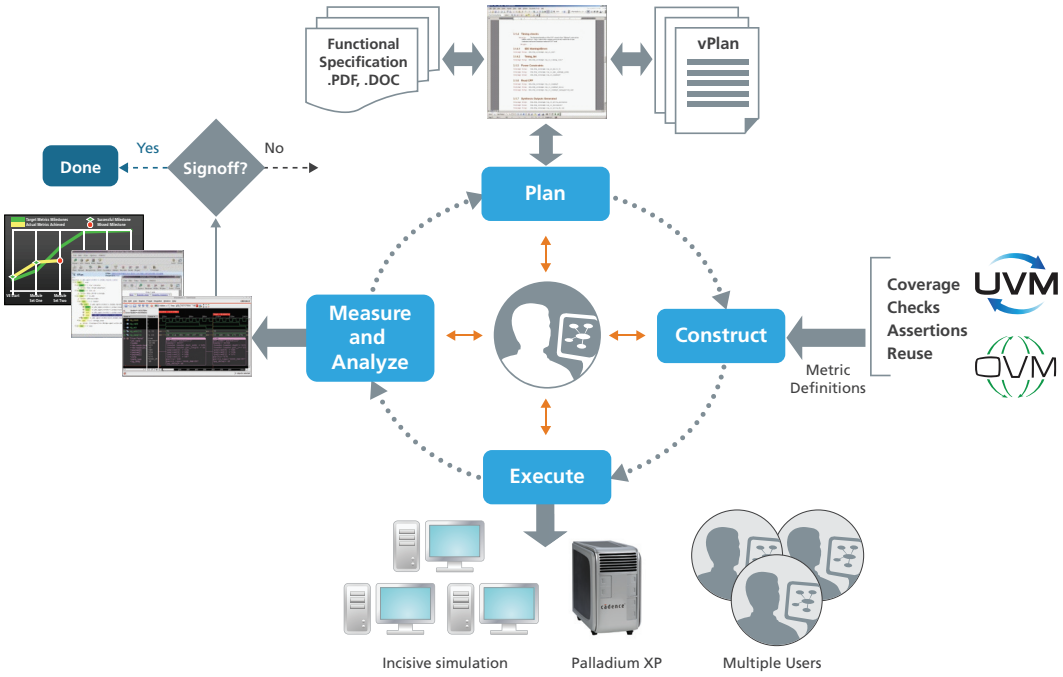


*Figure 6: Metric-driven verification flow*

## Summary

UVM/OVM acceleration extends the UVM/OVM to enable acceleration-ready UVCs and a metric-driven verification flow. It is a true UVM/OVM solution that utilizes the configuration mechanism, standard control knobs, high-level encapsulation, dynamic build process, and other UVM/OVM characteristics. UVM/OVM acceleration suggests a consistent, easy-to-follow use model for accelerated and simulated environments. It puts more focus on reuse and increases automation within provided library extensions. On top of the standard SCI-ME 2.0, standard usage offers a generic C package that relieves users from coding in C and facilitates an efficient solution for passing static- and dynamic-sized messages between the simulated and accelerated testbench parts.