Universal Verification Methodology for Mixed-Signal Standard (UVM-MS) Version 1.0

January 2025

Abstract: The Universal Verification Methodology for Mixed-Signal (UVM-MS) is a comprehensive and unified analog/mixed-signal verification methodology based on the Universal Verification Methodology Standard (UVM, IEEE Std 1800.2) that improves analog/mixed-signal (AMS) and digital/mixed-signal (DMS) verification of integrated circuits and systems. This standard defines a framework that enables the creation of analog/mixed-signal verification components and test benches by extending digital-centric UVM classes and facilitating interaction between class-based and structural environments. The objective is to standardize methods for driving and monitoring mixed-signal nets within UVM. The reuse of proven verification components will in turn increase the productivity of verification teams and improve overall quality. The overall UVM-MS methodology is explored with supporting examples and use cases.

Keywords: agent, AMS, analog, class, DMS, interface, messaging, mixed-signal, proxy, RNM, SystemVerilog, Universal Verification Methodology, UVM, UVM-MS, verification

Notices

Suggestions for improvements to the UVM-MS Methodology are welcome. They should be posted to the UVM-MS Community Forum at:

https://forums.accellera.org/forum/60-uvm-mixed-signal/

The current Working Group web page is:

http://www.accellera.org/activities/working-groups/uvm-ms

Participants

The UVM Mixed-Signal Working Group (UVM-MS WG) is entity-based. At the time this standard was completed, the UVM-MS Working Group had the following active participants:

Tom Fitzpatrick, Siemens EDA, *Chair*
Tim Pylant, Cadence Design Systems, *Vice Chair*
Shalom Bresticker, Retired, *Technical Editor*

Cadence Design Systems, Inc.: Chandrashekar Chetput, Abhijit Madhu Kumar
Intel Corporation: Boon Chong Ang, J. David Rosen
NXP Semiconductors: Joachim Geishauser, Manish Patel
Qualcomm Incorporated: Bob Pau
Renesas Electronics Corp.: Peter Grove, Steven Holloway
Siemens EDA: Mina Louis Zaki
Synopsys, Inc.: Tushar Pandey, Msehli Slim, Weiming Zhu

At the time of standardization, the UVM-MS WG had the following eligible voters:

Cadence Design Systems, Inc.

Intel Corporation

NXP Semiconductors

Qualcomm Incorporated

Renesas Electronics Corp.

Siemens EDA

Synopsys, Inc.

1

Contents

1. Overview

1.1 Scope

The Universal Verification Methodology for Mixed-Signal Standard (UVM-MS) is a comprehensive and unified analog/mixed-signal verification methodology based on the Universal Verification Methodology (UVM, IEEE Std 1800.2™) that improves analog/mixed-signal (AMS) and digital/mixed-signal (DMS) verification of integrated circuits and systems. This framework enables the creation of mixed-signal verification components and test benches by extending digital-centric UVM classes and facilitating interaction between class-based and structural environments. The objective is to standardize methods for driving and monitoring mixed-signal nets within UVM. The reuse of proven verification components will in turn increase the productivity of verification teams and improve overall quality.

The overall UVM-MS methodology is explored with supporting examples and use cases.

1.2 Purpose

Recent trends in integrated circuit and system design have seen an increase in the level of interaction between digital logic and analog circuitry. As the level of interaction has increased, the overall level of complexity of SoCs (Systems on Chips) has also increased rapidly. Functional digital verification has evolved over time to meet the demands of increased complexity, enjoying the benefits of automated flows, metric-driven verification, and other well-defined and understood methodologies. Analog functional verification methodologies have not developed at the same pace. Bridging the gap between the analog and digital methodologies is critical to ensuring robust verification of mixed-signal integrated circuits. Verifying mixed-signal designs is challenging and requires a solution capable of dealing with the increased complexity and ever-reducing time-to-market requirements.

UVM-MS extends the use of UVM components and extensions thereof into the physical layer, enabling mixed-signal verification. The methodology presents a solution for the following requirements:

- Define analog behavior based on a set of parameters defined in a **uvm_sequence_item** and generate that analog signal using a *mixed-signal bridge* module;
- Measure the properties of the analog signal, return them to a UVM monitor, package those properties into a sequence item, and potentially collect coverage and/or check correctness;
- Set controls for pre-run configurations via constraints.

1.3 Approach

Mixed-signal simulations contain signals in both analog and digital domains. These signals may need to be driven or monitored by UVM components. However, UVM components can only directly interact with these signals digitally. While the conversion to analog signals could be handled using Verilog-AMS *connect modules*, for example, the analog signals can more naturally, consistently, and accurately be modeled in the analog domain. At the same time, mixed-signal simulations may be performed with different signal representations, so it is desirable to have a uniform interface between the signal (regardless of domain) and the UVM component with which it interacts. In order to provide this uniform interface, a UVM-MS testbench needs to include a *bridge* between a UVM-accessible API and the actual SystemVerilog, Verilog-AMS, or SPICE connection to the signals.

UVM-MS is an extension to UVM, so that users can build on their existing UVM infrastructure as much as possible when they need to incorporate mixed-signal designs into their system. A typical UVM testbench consists of a set of classes, each derived from the **uvm_component** base type, that are instantiated and connected to allow stimulus and response checking between the testbench and the DUT (Device Under Test). Each DUT interface is connected via a SystemVerilog *virtual interface* to a **uvm_agent**, which includes the following protocol-specific components to interact with the DUT:

- The *uvm_sequencer* arbitrates between *uvm_sequences* and sends a *uvm_sequence_item* to the *uvm_driver*.
- The *uvm_driver* gets the *uvm_sequence_item* from the *uvm_sequencer*, and uses the information contained therein to communicate with the DUT via the virtual interface. The *uvm_driver* may also receive responses from the DUT and pass them back via the sequencer to the originating *uvm_sequence*.
- The *uvm_monitor* recognizes behaviors on the virtual interface and encapsulates these into *uvm_sequence_item* objects to communicate them to other components in the environment, possibly including coverage collectors and/or scoreboards.

UVM-MS takes advantage of the existing UVM architecture and extends it to allow interactions between the UVM agent and the mixed-signal DUT. As outlined below, UVM-MS defines a separate *MS Bridge* module that connects to the virtual interface to interact directly with the UVM driver and monitor on one side and to the mixed-signal DUT on the other.

The *MS Bridge Core* implements the physical connection functions required by the UVM agent driver and monitor as well as data type conversions between the agent and the DUT I/O (Input/Output). Thus, UVM-MS eliminates any requirements for Verilog-AMS connect modules within the MS Bridge. The core is written in the language that allows best possible representation of the DUT I/O net abstraction. The *MS Bridge* also includes an *MS Proxy* class object (a handle to which is passed to the UVM driver and monitor) to provide an application programming interface (API) to allow the driver to call proxy class methods that control the bridge core.

Updating an existing digital-only UVM environment can then be accomplished by defining a new driver class, extended from the existing UVM driver, that includes a pointer to the proxy class object and possibly utilizes a new sequence item extended from the digital-only sequence item. The UVM factory allows for these new classes to be used without having to modify the existing UVM environment code. Similarly, the UVM test could replace the original sequence with a new sequence that uses the new sequence item to define a verification scenario targeted at the mixed-signal DUT. The rest of the UVM environment could be reused with minimal, if any, modifications.

If starting with an analog/mixed-signal DUT, the UVM environment can be defined from the beginning to rely on the *MS Bridge* and the proxy class to interact with the DUT. All other aspects of the environment, including the UVM test, would follow existing UVM guidelines, thus providing for a modular, reusable, constraint-driven verification environment similar to what digital verification engineers have used for many years.

1.4 Word usage

The word *shall* indicates mandatory requirements strictly to be followed in order to conform to the standard and from which no deviation is permitted (*shall* equals *is required to*).[1,2]

The word *should* indicates that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required (*should* equals *is recommended that*).

The word *may* is used to indicate a course of action permissible within the limits of the standard (*may* equals *is permitted to*).

The word *can* is used for statements of possibility and capability, whether material, physical, or causal (*can* equals *is able to*).

[1] The use of the word *must* is deprecated and cannot be used when stating mandatory requirements; *must* is used only to describe unavoidable situations.
[2] The use of *will* is deprecated and cannot be used when stating mandatory requirements; *will* is only used in statements of fact.

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

Accellera VAMS-2023, Verilog-AMS Language Reference Manual. [3]

IEEE Std 1800™, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language. [4,5]

IEEE Std 1800.2™, IEEE Standard for Universal Verification Methodology Language Reference Manual.

3. Definitions, acronyms and abbreviations

3.1 Definitions

For the purposes of this document, the following terms and definitions apply. The normative references in Clause 2 and the *IEEE Standards Dictionary Online* should be consulted for terms not defined in this clause. [6]

AMS: Analog/Mixed-Signal (AMS) simulation and verification refers to systems that can simulate and verify analog/mixed-signal designs using co-simulation of digital (*logic* or *real* data types) event-driven engines and analog (*electrical* data types) time-domain solvers ("AMS co-simulation").

Blocking, non-blocking: A *blocking* task call suspends execution until it completes. A *non-blocking* call returns immediately.

Bridge core: A module in the MS Bridge that performs data type conversion and manipulation between the UVM agent and the DUT. This can include modeling of passive and active analog components. It is the bridge core that directly connects to and drives the DUT.

DMS/RNM: Digital (or Discrete)/Mixed-Signal (DMS) simulation and verification refers to event-driven systems that can simulate and verify analog designs using discrete abstractions of the analog design. Such designs can use SystemVerilog *user-defined net types* (*UDNs*) and/or *real* variables to represent the analog intent. There is no standard on how discrete modeling of analog nets should be done, so the examples given here are just one method of implementation. Sometimes the term RNM (Real-Number Modeling) is used. The term DMS will be used to refer to both DMS and RNM within this document.

MS: Mixed-Signal (MS) simulation and verification refers to verification of a DUT that is made up of digital and analog blocks. These analog blocks could be transistor designs or a model abstraction utilizing DMS, AMS or a mixture. The focus of this standard is to support MS rather than specifically DMS or AMS.

OOMR: An Out-of-Module Reference (OOMR) is a reference from one module to another. Sometimes called XMR (Cross-Module Reference).

Proxy: A class functioning as an interface to another **component** or class.

UVM: The Universal Verification Methodology that enables creation of robust, reusable, modular and interoperable verification IP and testbench components for digital verification. UVM is documented in IEEE Std 1800.2.

[3] Accellera publications are available from the Accellera Systems Initiative (https://accellera.org/).
[4] IEEE publications are available from the Institute of Electrical and Electronics Engineers (https://standards.ieee.org/).
[5] The IEEE standards or products referred to in Clause 2 are trademarks owned by the Institute of Electrical and Electronics Engineers, Incorporated.
[6] *IEEE Standards Dictionary Online* is available at: http://dictionary.ieee.org. An IEEE Account is required for access to the dictionary, and one can be created at no charge on the dictionary sign-in page.

UVM-MS: A comprehensive and unified mixed-signal verification methodology based on UVM to improve AMS and DMS verification of integrated circuits and systems.

3.2 Other acronyms and abbreviations

API	Application Programming Interface
DC	Direct Current
DUT	Device Under Test
IF	Interface
I/O	Input/Output
IP	Intellectual Property
PWL	Piecewise-Linear
SV	SystemVerilog
UDN	User-Defined Nettype

4. UVM-MS architecture

4.1 Introduction

The structure of a generalized UVM-MS agent and its connection to a mixed-signal DUT are shown in Figure 1 below. A UVM-MS agent differs from a digital-only UVM agent in that it contains a handle to a proxy class in addition to the virtual interface to communicate with the DUT. The *MS Bridge* is the layer between the agent and the mixed-signal DUT.

The *MS Bridge* is a SystemVerilog module comprising a proxy class object (the *MS Proxy*), a SystemVerilog interface (the *SV IF*), and a *Bridge Core* module. The bridge core performs signal data type conversion and manipulation between the UVM agent and the DUT. This can include modeling of passive and active analog components. It is the bridge core that directly connects to and drives the DUT. The proxy class provides an API that conveys analog attributes between the agent and the bridge core. The SV IF is used only for passing logic-type (i.e., not *real* or *user-defined nettype* values) between the agent and the bridge core, whereas the proxy is used to drive values for controlling analog signal generation or to monitor aspects of the analog signal.

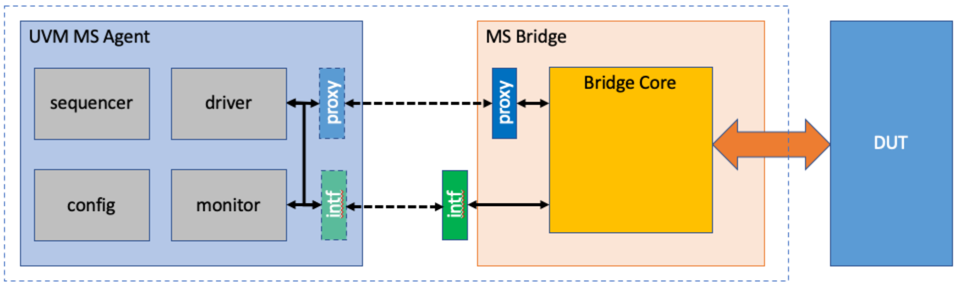


Figure 1: UVM-MS Architecture

The agent contains a handle to the MS Proxy and a virtual interface to the SV IF. The dashed lines in the diagram indicate communication via these mechanisms. The MS Proxy and SV IF can be used together or individually, depending on the types of connections required. The SV IF may be placed either within the MS Bridge or between the agent and the bridge. This document presents only one of those options in most places, but unless stated otherwise, both are possible. See Clause 5 for more details.

The bridge core can be modeled with SystemVerilog, Verilog, or Verilog-AMS, based on the net types of the DUT I/O it connects to. The choice of the modeling abstraction for the bridge core is made at elaboration time. The bridge may contain more than one bridge core, of the same or different types. Each core may connect to one or more SV interfaces. However, a bridge contains only one proxy, regardless of the number of cores inside.

As mentioned above, some changes are necessary to UVM drivers, monitors, and other objects to accommodate the handle to the proxy class and use it appropriately. If a testbench already exists, this can be done by deriving a mixed-signal enhanced version of the digital component and using UVM factory overrides to replace the originals. Then a modest set of changes can add the critical capabilities without disrupting any other purely digital applications of the UVM collateral.

4.2 MS bridge

UVM-MS introduces the concept of an *MS Bridge*, whose sole purpose is to create a bridge between the UVM-MS agent and the DUT, since direct connection via an SV interface is not possible for all DUT I/O abstractions. An example of a bridge is shown below in Figure 2. It uses the analog attributes from the proxy to generate continuously changing values (e.g., ramping voltage supply, electrically modeling drive strengths, R/C loading, etc.) in the bridge core.

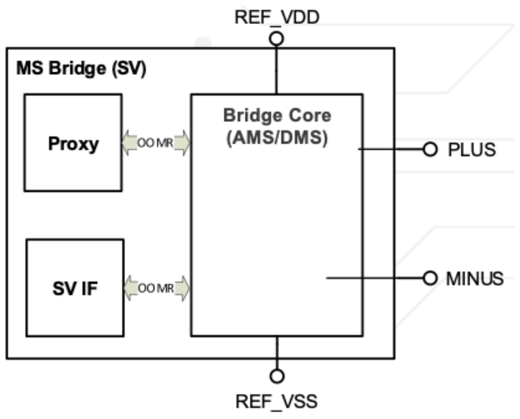


Figure 2: Example MS Bridge

The MS Bridge is written in SystemVerilog and shall contain the bridge core instance. The bridge core may use SystemVerilog or Verilog-AMS data types to drive the DUT pins. The bridge core abstraction can be changed and the existing UVM-MS framework will not require any other changes as long as the API is kept the same.

Although different abstractions of the bridge core can be used, Verilog-AMS only supports a limited set of port data types, so the SV interface and the proxy shall connect to the bridge core using OOMRs, to avoid these restrictions. These OOMRs shall only refer to constructs found in Verilog-AMS, which are common to all abstractions (but less limited than Verilog-AMS port data types).

The DUT I/O of the MS Bridge shall be defined as SystemVerilog `interconnect` types. The net type of the bridge DUT I/O should be resolved from the types of its connections, which may vary with the model abstraction level, so defining ports as `interconnect`, which is typeless, allows the simulator to do this and also prevents OOMRs and probes to them. Unless it is certain that the direction will always be unidirectional (`input` or `output`), it is recommended that the direction be declared as `inout`, which is more precise and is portable with electrical modeling.

While the MS Bridge is written in SystemVerilog, the DUT ports and the Bridge Core ports may be Verilog-AMS data types that are not supported in SystemVerilog. In order to support UVM-MS, the simulation tool needs to support connections of the Bridge Core to the DUT of such data types through the MS Bridge I/O.

The MS bridge ports and the bridge core ports to the DUT shall not be unpacked arrays, which are not supported by Verilog-AMS. For the same reason, ports may have only one packed array dimension.

4.2.1 MS proxy

In UVM, SystemVerilog *interfaces* are used to allow communication between the class-based portion of the testbench and the DUT. There are several shortcomings with interfaces that impact mixed-signal verification:

- Setting parameters on an interface (e.g., for analog component values) results in a specific data type that ripples through UVM configuration, hampering reuse.
- Implementing a bridge core API in an interface limits reuse, as it is impossible to override methods.
- It is not possible to communicate with Verilog-AMS code via an interface.

Due to these shortcomings, an alternative approach, the *MS Proxy*, is used. The MS Proxy in the MS Bridge follows the abstract/concrete class design pattern. The **uvm_ms_proxy** definition is provided within the **uvm_ms_pkg.sv** file (see A.1). An abstract extension of **uvm_ms_proxy** is then declared with prototypes of a set of methods that form the API through which a UVM-MS agent communicates with the MS Bridge.

Example:

```
import uvm_ms_pkg::*;
virtual class vdriver_proxy extends uvm_ms_proxy;
  pure virtual function void set_voltage(…);
  pure virtual function real get_voltage(…);
  pure virtual function void push(…);
  pure virtual function void pull(…);
. . .
endclass: vdriver_proxy
```

The concrete version of the proxy class is declared in the MS Bridge and extends the abstract version, implementing the API methods. A handle to the concrete proxy class is assigned to an abstract class handle in a UVM agent using the **uvm_config_db**. Polymorphism allows the UVM agent to handle different implementations of the abstract class without modification, e.g., implementations to suit DMS/AMS models of bridge cores.

Example:

```
module vdriver_bridge(PLUS, MINUS);
...
  vdriver_core #(…) i_core (…); // MS model

  class MSproxy extends vdriver_proxy;
    virtual function void set_voltage(…);
    ...
    endfunction: set_voltage
    ...
  endclass: MSproxy

  MSProxy i_proxy = new("i_proxy");
...
endmodule: vdriver_bridge

module hw_top;
  ...
  vdriver_bridge  i_vdriver_bridge (.PLUS(vcc),.MINUS(gnd));
  ...
endmodule: hw_top
```

```
class tb extends uvm_env;
  ...
  function void build_phase(uvm_phase phase);
    uvm_config_db#(vdriver_proxy)::set(null, "uvm_test_top.env", "vdriver_proxy",
       i_driver_bridge.i_proxy);
    ...
  endfunction
  ...
endclass: tb
```

A typical MS Proxy API contains methods to implement the following features, described in more detail in Clause 6:

- **Pull** (read) bridge core values via function calls to the bridge core.
- **Push** (write) bridge core attributes using function calls to the bridge core.
- **Push-Sync** contains registers for end-of-transition detection or other synchronization from bridge core control.
- **Monitor** continuous signals.

4.2.2 SV interface

The existing SV interface can continue to be used for logic-type communication normally used for UVM, thus enabling reuse of existing verification collateral. There is a limitation on bidirectional signals, which requires the driver and receiver part of the net to be manually split.

The SV interface can be located inside the MS bridge or outside the bridge, between the UVM agent and the bridge. See Figure 3 and Figure 4. Each method has certain advantages.

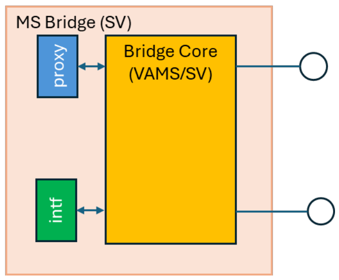


Figure 3: SV interface inside bridge

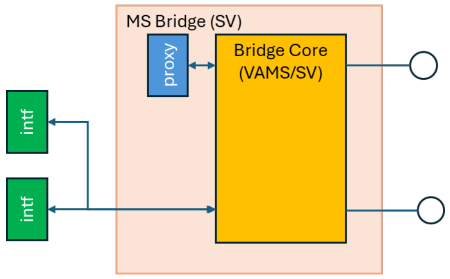


Figure 4: SV interface outside bridge

Example code of an SV interface outside the bridge:

```
interface my_intf;
...
endinterface

module tb;

   my_intf   i1();
   ms_bridge i2(.my_if(i1),.PLUS.....);

   // + DUT instantiation

endmodule

module ms_bridge(inout interconnect PLUS, ..., interface my_if);

   // SV IF to Core connections that are bidirectional
   my_short i1(my_if.a, core.net1);
   my_short i2(my_if.b, core.net2);

   // + Proxy declaration and instantiation
   // + Core instance

endmodule
```

The SV interface is discussed in more detail in Clause 5.

4.2.3 Bridge core

The MS testbench may require the behavior and presence of passive and active analog components that a typical UVM-RTL testbench could not include, such as:

- Capacitors, resistors, inductors, diodes, current/voltage sources, etc.
- A complex passive network
- A piece of Verilog-AMS code

Such components will be used to model the analog behavior of pads, lossy transmission lines, loads/impedances, voltage/current-controlled sources, etc., required to accurately model the signals connecting to the ports of the DUT. Those components can be placed inside the bridge core, to be controlled by the proxy.

In Figure 2, the example bridge core has four pins. PLUS and MINUS are two typical terminals on a component, which could be a voltage/current source or a capacitor, for example. REF_VDD and REF_VSS allow a logic input from the SV IF to be converted to a voltage relative to these values. In the MS Bridge, these ports are declared as **`inout interconnect`**, so the actual net types are determined by their connections, in this case, the bridge core ports and the DUT ports.

In AMS, the bridge core could instantiate SPICE primitives or Verilog-A modules[7]; however, this restricts pre-simulation changes in component values that are possible during the UVM pre-run phases if bridge core functionality is instead written using Verilog-A or Verilog-AMS analog contribution statements.

In DMS, there are no restrictions. Verilog-A modules can be compiled as Verilog-AMS, and the predefined Verilog-AMS macro **`` `__VAMS_ENABLE__ ``** allows Verilog-AMS-only code to be excluded when the same file is parsed as Verilog-A. An example where a SPICE primitive might be needed is when a printed circuit board component is

[7] Accellera VAMS-2023, Verilog-AMS Language Reference Manual, Annex E.

supplied as a SPICE model. It is recommended to use Verilog-AMS analog statements to implement the electrical functionality whenever possible.

The SV interface and the proxy shall connect to the bridge core with OOMRs, because Verilog-AMS only supports a limited set of port data types and does not allow unpacked array ports. The SV interface may use direct OOMRs to variables or wires in the bridge core, but the proxy should call functions defined in the bridge core. An example of when a function cannot be used is when a bus of reals needs to be passed into the bridge core, as Verilog-AMS does not support this. Another exception is shown in 6.1.5, where a bridge core synchronization variable is directly assigned to a proxy variable.

4.2.3.1 Driving continuous analog signals

Perhaps the major difference between digital and analog simulations is that digital signals tend to change discretely, whereas analog signals may be continuously changing. Many useful analog signals can be composed of other periodic waveforms, such as a series of sine waves. Taken to the extreme, a square wave, which is modeled digitally as "set to 1; wait a short time; set to 0; wait a short time," etc., could actually be modeled as an infinite series of harmonic sine waves, although it would never actually be implemented that way.

In practice, a sine wave can be controlled simply by four values defining the frequency, phase, amplitude, and DC bias of the generated signal. A UVM transaction can be defined that encodes the properties of the desired sine wave as real values in the **uvm_sequence_item**. This item can then be passed to the UVM driver, which will in turn pass the values to the MS Bridge Core via the MS Proxy. The Bridge Core will use those values to generate the signal.

In general, to drive such a continuous signal, the **uvm_sequence_item** would contain fields for all desired control parameters of the desired analog signal, and the driver will drive those settings through the proxy class to the MS Bridge Core to control the signal generator, which should be located in the domain of the DUT I/O it will connect to. For example, an analog sine wave should be implemented in the analog domain although controlled from the digital domain.

In this way, the UVM paradigm of having a relatively simple interface for the test writer is continued, providing a way to control the generated signal as needed.

4.2.3.2 Multiple signal generators

It is possible to combine multiple signal generators to create more complex signals. For example, a signal source may be defined to be a sine wave, as described in the previous section. An additional requirement may be to inject some noise onto that signal. There are several ways to implement this. The noise source may use a different **uvm_sequence_item** and may be controlled as part of the main UVM sequence, which could be used to inject noise at predictable intervals within the sequence. Alternatively, it may be generated by a different sequence, where the noise might be injected randomly. In either case, the driver will detect the sequence items as either signal or noise and pass the appropriate information on to the signal generator(s), depending on the environment.

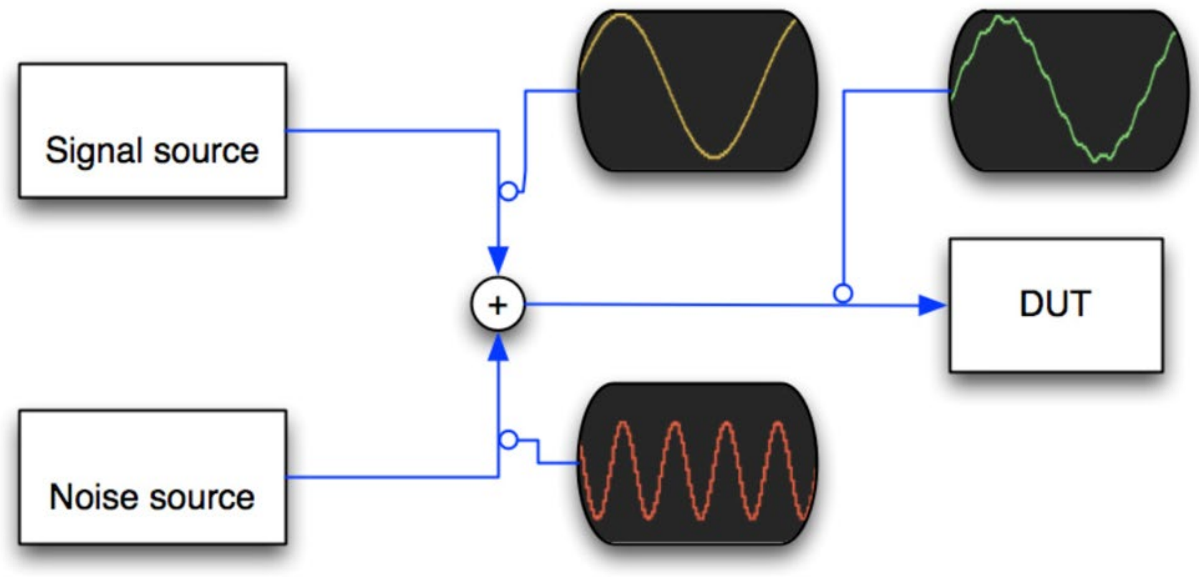


Figure 5: Multiple analog drivers on a single net

4.3 UVM-MS agent

In general, adapting a pure digital UVM agent for UVM-MS potentially requires extensions and factory overrides of the agent, driver, monitor, and sequence item classes.

The UVM-MS agent is in effect no different from a **uvm_agent**. However, some design choices can simplify extension to allow operation with an MS DUT. For example, the handles to the MS proxy class and the virtual interface can be contained in the agent configuration object. If there is an existing agent, it can be extended to create the MS agent.

Example:

```
class osc_ms_agent extends osc_agent;
```

4.3.1 UVM-MS driver

The UVM-MS driver is responsible for requesting and driving transactions from/to the agent sequencer. Driving can involve communication with an MS proxy and/or an SV interface. Patterns for implementing the communication are described in Clause 6. If there is an existing driver, it can be extended to create the MS driver.

Example of a driver call to a *push* method defined in the proxy:

```
virtual task drive_transaction(osc_ms_transaction req);
  ...
  bridge_proxy.push(.ampl(req.ampl),.bias(req.bias),.freq(req.freq));
  ...
```

4.3.2 UVM-MS monitor

A UVM-MS monitor extends the UVM monitor by adding a handle to the MS Proxy. The MS Proxy supports DMS/AMS DUT connections by implementing *pull* and *monitor* methods, as described in Clause 6. Logic signals transferred via the SV IF can be monitored using the existing methods in UVM. If there is an existing monitor, it can be extended to create the MS monitor.

Example:

```
class osc_ms_monitor extends osc_monitor;
```

4.3.3 UVM-MS sequence item

The sequence item used by the UVM-MS agent will likely have additional fields that will be used by the MS proxy.

Example:

```
class osc_ms_transaction extends osc_transaction;
  rand real ampl;
  rand real bias;
  ...
```

4.4 UVM-MS scoreboard and coverage

Scoreboard and coverage architecture may not need to change. However, they may need to be extended to add additional checks or to take into account additional information found in the extended transaction type.

Example:

```
class osc_ms_scoreboard extends osc_scoreboard;
```

4.5 UVM-MS testbench

The testbench needs to be extended to add the **uvm_config_db** call to pass the proxy handle and the **set_override** calls to specify the overrides for any scoreboard, agent, driver, monitor, and sequence item that have been extended.

The following example is excerpted from the frequency adapter example in Annex B:

```
class freq_adpt_ms_tb extends freq_adpt_tb;

  `uvm_component_utils(freq_adpt_ms_tb)

  function new (string name, uvm_component parent=null);
    super.new(name, parent);
  endfunction : new

  // UVM build() phase
  virtual function void build_phase(uvm_phase phase);

    // set up bridge proxy pointer references in generator & detector agents
    uvm_config_db #(osc_proxy)::set(this,"freq_generator.agent.*",
      "bridge_proxy", top.generator_bridge.proxy);
    uvm_config_db #(osc_proxy)::set(this,"freq_detector.agent.*",
      "bridge_proxy", top.detector_bridge.proxy);

    // override driver, monitor, and scoreboard with UVM-MS versions
    set_type_override_by_type(osc_transaction        ::get_type(),
                              osc_ms_transaction     ::get_type());
    set_type_override_by_type(osc_driver             ::get_type(),
                              osc_ms_driver          ::get_type());
    set_type_override_by_type(osc_monitor            ::get_type(),
                              osc_ms_monitor         ::get_type());
    set_type_override_by_type(freq_adpt_scoreboard   ::get_type(),
                              freq_adpt_ms_scoreboard::get_type());
    super.build_phase(phase);

  endfunction : build_phase

endclass: freq_adpt_ms_tb
```

5. Bridge configuration

The MS Bridge DUT I/O ports shall be of type **`interconnect`**, which disallows probes or assignments to these ports at this level. All probes and assignments should be done in the bridge core that is instantiated in the MS Bridge. Declaring the port as the typeless **`interconnect`** not only prevents OOMRs, but also means that the net type will be resolved from its connections. It is recommended that the direction be **`inout`**.

Classical UVM has SystemVerilog interfaces instanced along with the DUT in a module. The DUT ports can be connected to an instanced interface as individual port connections or themselves be grouped as a port of **`interface`** type. If the DUT ports are an interface, then for UVM-MS, these shall only contain logic-type data types and do not require a MS Bridge, as these ports cannot have any other abstraction.

UVM-MS supports the SV IF instanced inside the MS Bridge or external to the bridge. The SV IF shall only contain integral signals.

If the SV IF is external to the MS Bridge (Figure 6), this allows multiple types of interfaces to be hooked up, e.g., in a scenario where the DUT I/O could be used for multiple digital communication functions. For example, a group of pins might work with I2C or with SPI. This allows the MS Bridge to be a generic connector decoupled from the digital side.

In this configuration, the SV IFs connect to MS Bridge ports. Within the bridge, these ports connect to the bridge core with OOMRs, not with ports, just as the MS Proxy connects to the core with OOMRs. This configuration enables the insertion of the MS Bridge between the DUT and SV IF as a simple step from UVM to UVM-MS. However, the hierarchical path to the SV interfaces is then different from the path to the proxy and the bridge.

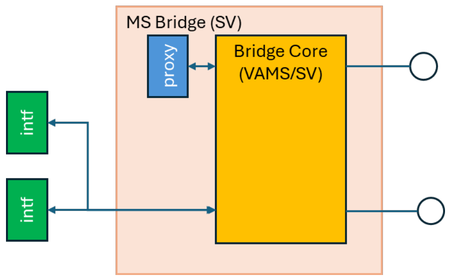


Figure 6: SV interface connection external to bridge

If the SV IF is internal to the MS Bridge (Figure 7), an external IP provider only has to supply a single module to instantiate with the DUT. This has the advantage that encryption is easier and the hierarchical paths to the proxy and the interface are the same.

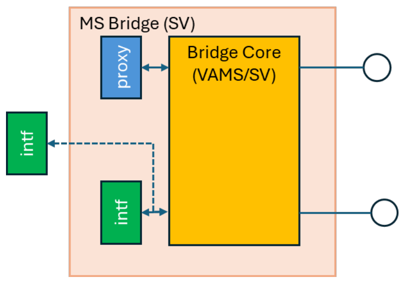


Figure 7: SV interface connection internal to bridge

Table 1 below shows the various possible configurations of the MS Bridge. In each case. the SV IF is shown within the MS Bridge, but as discussed above, it could be external to the bridge.

The table describes the configuration assuming a single DUT port abstraction type. However, an MS Bridge could have multiple types of DUT I/O connections that make up a protocol, e.g., a mixture of multiple supplies and logic signals, such as USB. A bridge could also have more than one core internally.

In the table, DUT I/O type abstractions are divided into two kinds: *digital,* whose DUT I/O abstraction is made up of 2-state or 4-state *logic* types, whether individually or as part of an SV interface port, and *mixed-signal,* whose abstraction can use *real*s or *wreal*s (RNM), *UDN*s (DMS) or *electrical* (AMS). A subtype of mixed-signal is when the port is a pure analog *passive network.*

A *Passive network* refers to the use case where the bridge controls an analog passive network in the bridge core needed for the DUT, such as a speaker model. The analog passive network can be made up of many controllable components, such as L/R/Cs. An example is a resistor used as a switch or variable resistance between two DUT pins.

Mixed-signal types can contain, for example, multiple sources of stimulus, such as controlling the data type conversion of the interface to the DUT I/O (slew rate, A2D/D2A logic levels, output drive impedance, etc.) and driving a voltage or current with some waveform representation (see Figure 8).

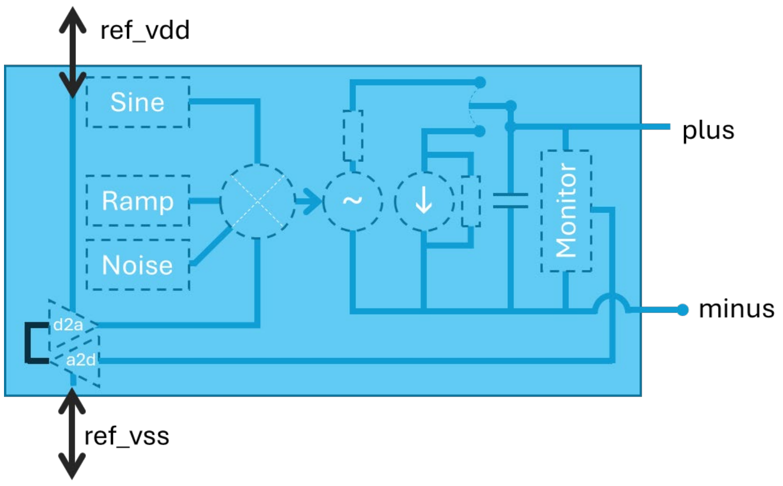
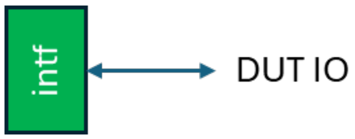
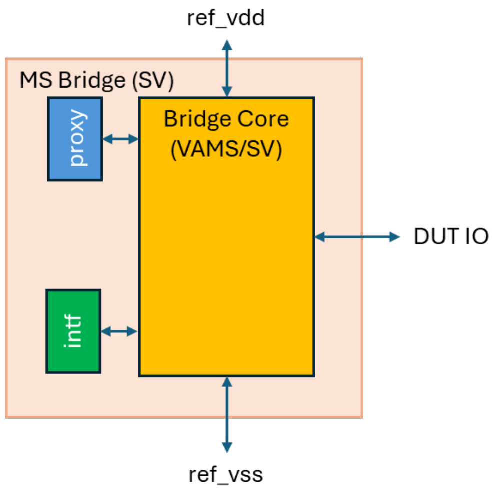
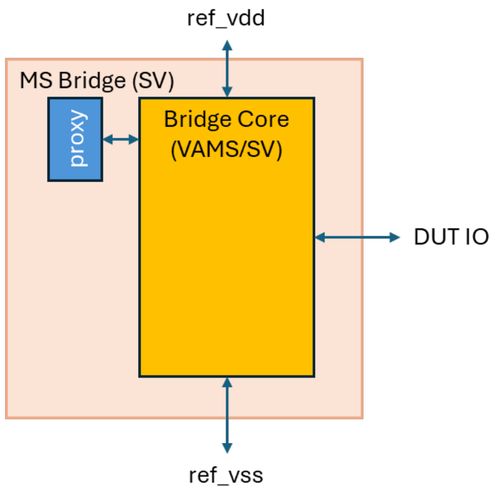


Figure 8: Mixed-signal bridge

Table 1: Bridge configurations

Diagram	Description
Option 1 	**Digital**: Supported **Mixed-signal**: Not supported **Passive network**: Not supported **Notes**: Here, in all DUT abstractions, its I/O is logic (discrete) or an SV interface port. DUT I/O is always digital, so it is a classic UVM connection. UVM-MS stipulates that no connect modules should be used in the UVM-MS framework, so this configuration cannot be used in mixed-signal, as different abstractions of the DUT I/O would not be supported. To support a mixed-signal DUT I/O, a MS Bridge is required.
Option 2  ref_vdd MS Bridge (SV) proxy Bridge Core (VAMS/SV) DUT IO intf ref_vss	**Digital**: Supported **Mixed-signal**: Supported **Passive network**: Not supported **Notes**: The bridge core abstraction is changed based on the DUT I/O data type. Easily supports DUT I/O of logic, real, UDN, or electrical type. The proxy is used to control the data conversion from the interface data type to the DUT data type if different, as well as being used to control a MS passive or active component. This is the most configurable connection to a DUT I/O that has many different functional uses. Examples of bridge core abstractions; • Short-circuit between OOMR from SV IF to bridge core to DUT I/O, if DUT I/O is digital, Mixed-signal cannot be represented in this case. (SV view) • DMS/RNM converter for the interface path to the DUT I/O, in addition to mixed-signal features. (SV view) • AMS converter for the interface path to the DUT I/O, in addition to mixed-signal features. (Verilog-AMS view)
Option 3  ref_vdd MS Bridge (SV) proxy Bridge Core (VAMS/SV) DUT IO ref_vss	**Digital**: Not supported **Mixed-signal**: Supported **Passive network**: Supported **Notes**: The proxy is used for push/pull commands to bridge core. Examples of the bridge core abstractions: • Open if DUT I/O cannot represent a passive load. (SV view) • DMS/RNM representation of the passive load. (SV view) • AMS representation of the passive load. (Verilog-AMS view)

Passive analog components, such as capacitors, resistors and inductors, often have their value set once and left unchanged during the simulation. A typical setup would use parameters on the instance to set their default values. This makes the values easier to review. To avoid coding the default values for these components in the agent, functions are used to read the parameter values and to store them in the **uvm_config_db**. In the following example, the **get_parameters()** function in the proxy is used to read all the parameters and pass them back to the agent to apply them to the **uvm_config_db** in the UVM *connect* phase. If a testcase requires a change, the values are changed by a call to **set_parameters()** in the UVM *start_of_simulation* phase, before time is consumed.

The bridge core has internal variables that are initialized to the parameter values, but can be later overridden by a **set_parameters()** function call (recommended) or by an OOMR. These variables are the values that are used in the subsequent code. The **get_parameters()** function reads the initial values of the bridge core variables to ensure that any **`defparam`** statement override is considered. It is recommended that the configuration be written out, as a mistake in the values could be hard to debug.

Example:

```
package res_pkg; // This package contains definitions of common variables and types

  class res_config extends uvm_object;
    real res_val, res_tr, res_tf;

    `uvm_object_utils(res_config)

    function new(string name = "res_config");
      super.new(name);
    endfunction : new
  endclass: res_config

  virtual class res_proxy extends uvm_ms_proxy;
    pure virtual function void set_parameters(res_config cfg);
    pure virtual function res_config get_parameters();
    . . .
  endclass: res_proxy

endpackage: res_pkg


module ms_bridge #(parameter real res_val=1.0, res_tf=1.0e-9, res_tf=1.0e-9)
                  (inout interconnect PLUS, MINUS);

import uvm_pkg::*, uvm_ms_pkg::*;

`include "uvm_macros.svh"
`include "uvm_ms_includes.svh"

import res_pkg::*;

//================================================================
// API implementation of the proxy that the MS agent connects to
//================================================================
class MSproxy extends res_proxy;

    function new(string name);
      super.new(name);
    endfunction: new

    //======================================================
    // Function to get initial values for config db taken
    // from bridge core in case a defparam has been used
    //======================================================
    virtual function res_config get_parameters();
        res_config cfg = new();
        cfg.res_val  = i_core.rseries_val;
        cfg.res_tr   = i_core.rseries_tr;
        cfg.res_tf   = i_core.rseries_tf;
        return(cfg);
    endfunction: get_parameters

    //==================================================
    // Function to set initial values from config class
    // into bridge core after modification by uvm_test
    //==================================================
    virtual function void set_parameters(res_config cfg);
        i_core.rseries_val  = cfg.res_val;
        i_core.rseries_tr   = cfg.res_tr;
        i_core.rseries_tf   = cfg.res_tf;
    endfunction: set_parameters

    // Other code
endclass: MSproxy


// Instantiate proxy class
MSproxy i_proxy = new("i_proxy");

// instance of bridge core
bridge_core i_core #(.res_val(res_val),
                     .res_tr (res_tr) ,
                     .res_tf (res_tf))
                    (.PLUS (PLUS),.MINUS(MINUS));
endmodule: ms_bridge
```

Bridge Core:

```
`include "constants.vams"
`include "disciplines.vams"

module bridge_core (PLUS, MINUS);
   inout          PLUS, MINUS;
   electrical     PLUS, MINUS;

   // Values read by get_parameters() in MS Bridge
   parameter real res_val = 1.0;
   parameter real res_tr  = 1.0e-9;
   parameter real res_tf  = 1.0e-9;

   // Initial values set from parameters, then overriden by set_parameters()
   // Code uses rseries_* variables
   real rseries_val = res_val;
   real rseries_tr  = res_tr;
   real rseries_tf  = res_tf;

endmodule
```

6. Bridge core communication

The goal of the bridge core communication is that it should work whatever the abstraction of the bridge core is. The Proxy/SV interface connection to the bridge core needs to be the same for any bridge core abstraction, so that the abstraction can be changed without affecting the rest of the agent and the MS Bridge. Thus, any abstraction needs to implement the same functions. This also leads to the code having to conform to the subset that is common to both SystemVerilog and Verilog-AMS.

Therefore, the **uvm_hdl_*** methods shall not be used to read or write the logic variables.

Section 6.1 discusses communication in an AMS environment and section 6.2 describes communication in DMS.

6.1 AMS communication

Verilog-AMS defines ways in which continuous analog domain signals can update values that are then used in the digital domain. The Verilog-AMS system functions to do this are **`cross()`**, **`above()`**, and **`absdelta()`**, which can be used in a digital **`@()`** statement (**`above()`** is strongly preferred over **`cross()`**). This portion is the A->D trigger direction, in that the analog domain will generate the event for the digital domain to use. Use cases are:

- Event generation from analog signals (**`cross()`**, **`above()`**, **`absdelta()`**)
- Quantized continuously monitored analog signals (**`absdelta()`**)

Verilog-AMS defines how the discrete domain can read analog values and how digital inputs shall be interpreted by the analog domain. Digital inputs that are read by the analog domain should be filtered by a **`transition()`** function to smooth out the discrete change into a PWL signal. This portion is the D->A trigger direction, in that the discrete domain will generate the event for the analog to then use. These can be categorized into the following use cases;

- **Pull** (read) analog values via event-driven calls (do not cause an analog timestep, and values are interpolated)
- **Push** (write) bridge core controls
- **Push** bridge core controls with handshake synchronization

Combinations of A->D and D->A triggers can be used, e.g., an A->D event being used to pull an analog value causes a D->A event. The subtleties are discussed in the next section to ensure robust communication. The UVM-MS methodology does not dictate how this is done, but points to possible “gotchas” that are often missed.

6.1.1 Event generation from analog signals

Events in the digital domain can be generated from analog signals in two ways. In the first way, a **`cross()`** or **`above()`** function call can be used to generate a digital event when an analog signal meets the requirements for the function to trigger the event. This method is useful, for example, when the exact point the signal goes out of range is required, but at a performance cost.

Example:

```
always @(above(V(in)−0.9,…) or above(0.5-V(in),…)) -> out_of_range;
```

The second way is when the nearest analog timestep is good enough, in which case the following code is suitable. The **`absdelta()`** function is used to sample the analog value `out_of_range_ana` when its value toggles between 0 and 1. The advantage of this approach is that it does not affect the analog time step algorithm.

```
integer out_of_range_ana;
analog begin
  if((V(in) >= 0.9) || (V(in)=<0.5)) out_of_range_ana = 1;
  else                               out_of_range_ana = 0;
end
always @(absdelta(out_of_range_ana,1,0,0,1)) -> out_of_range;
```

NOTE—In Verilog-AMS, an event occurs whenever a value is assigned to an analog variable that appears in a **`@()`** event control, regardless of whether the variable changes value or not, so triggering off `out_of_range_ana` directly would cause an event at every analog timestep regardless of whether the value changed or not.

6.1.2 Quantized continuously monitored analog signals

Verilog-AMS provides the function **absdelta()** to quantize an analog signal for use in the discrete domain. The **absdelta()** function provides controls how to do the quantization as well as an enable. In addition, it is possible to query the analog value from the discrete domain by an event, such as a periodic clock. The choice between **absdelta()** and a periodic sampling depends on the use case and on the signal to achieve optimal simulation performance. Some examples are shown below; there is no generic solution that can be applied. See the Verilog-AMS standard for a comprehensive description of the **absdelta()** function.

Setting the *delta*, *time_tol* and *expr_tol* arguments appropriately allows **absdelta()** to operate accurately. Using a periodic digital event to sample the value may seem simpler, but each query causes a request to the analog solver to interpolate the *analog_expr* to get a value for that digital time point. Each request for the analog value is a blocking request, so control passes to the analog solver to get the value before returning to the digital solver.

```
always @(absdelta(analog_expr, delta, time_tol, expr_tol, enable))
  discrete_real = analog_expr;

forever #10              discrete_real = analog_expr;
always @(posedge clk)    discrete_real = analog_expr;
```

6.1.3 Event-driven queries of analog values

It is possible to use a digital function within a Verilog-AMS file to probe continuous variables or nets. The returned value is interpolated from the last accepted matrix solution and the next proposed solution. The proxy would use a function call to the function in the bridge core to return the value. (Note that functions in Verilog-AMS are required to have at least one input, even if not needed, and support only a limited set of data types.)

```
function real get_voltage(input dummy);
   begin
       get_voltage = V(PLUS);
   end
endfunction

function real get_current(input dummy);
   begin
       get_current = I(<PLUS>);
   end
endfunction
```

6.1.4 Push analog control

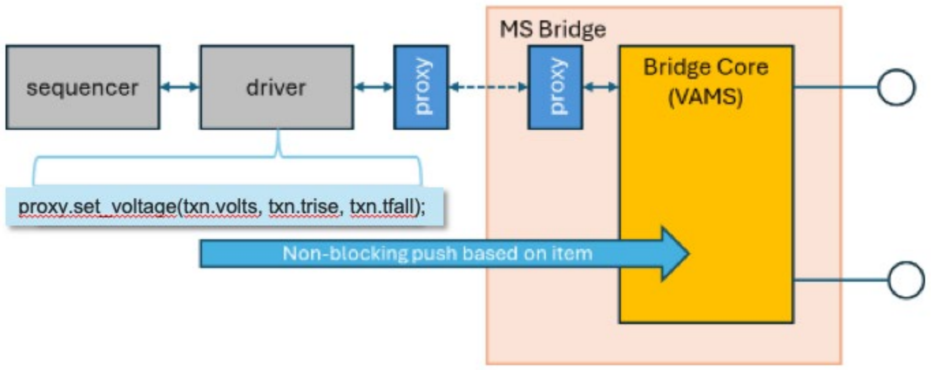
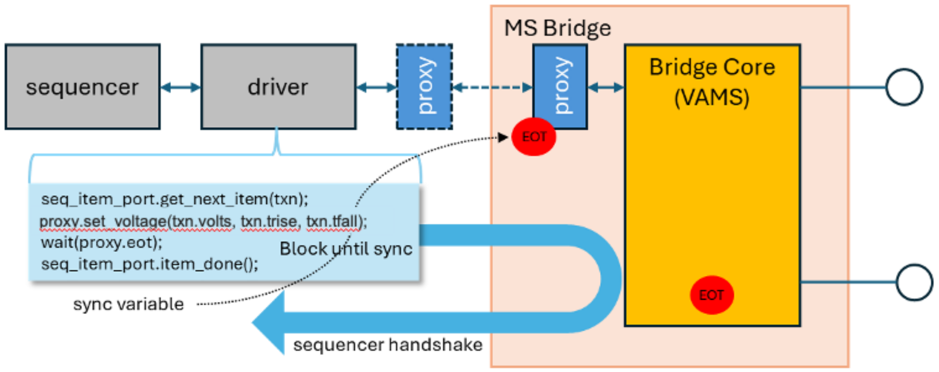

MS Bridge
sequencer
driver
proxy
proxy
Bridge Core (VAMS)
proxy.set_voltage(txn.volts, txn.trise, txn.tfall);
Non-blocking push based on item

Figure 9: Push pattern for analog control example

The **push** pattern for analog control makes an instantaneous change to discrete bridge core variables and does not wait for an event or a signal change to happen before returning. The agent driver simply calls a function in the proxy class that in turn either calls a function in the bridge core or uses an OOMR to set the value of a variable internal to the bridge core. The proxy function then immediately returns control to the driver. In Figure 9, `txn` is a transaction instance name.

6.1.5 Push analog control with handshake (push-sync)



```
seq_item_port.get_next_item(txn);
proxy.set_voltage(txn.volts, txn.trise, txn.tfall);
wait(proxy.eot);
seq_item_port.item_done();
```

Figure 10: Push-sync pattern for analog control example

The **push-sync** mechanism synchronizes the digital and analog simulation engines, since the analog solver requires time to change signal values. This pattern drives a change to the bridge core and waits for an event based on a signal condition before passing control to the next sequence item. This can be useful for such operations as voltage ramping, etc.

NOTE (informative): The example shown in Figure 10 uses wait(proxy.eot) rather than a delay that matches the transition time to allow for potential tolerances in the analog solver that may cause the final value to be reached slightly before of after that actual transition time specified. In addition, the set_voltage() call used in the example could be a combination of other changes in the Bridge Core that monitors for completion.

A push-sync pattern implementation shall declare a synchronization variable in the bridge core that indicates when the event has occurred, and a corresponding synchronization variable in the proxy class. The driver waits for the proxy synchronization variable before continuing.

Use function calls to prevent race conditions. A function does not return until its nested function calls have also completed.

For example, the transition filter **`transition()`** in Verilog-AMS smooths a discrete signal into a continuous-time signal. It stretches instantaneous changes in signals over a finite amount of time and can delay the transitions, as shown in the following figure.

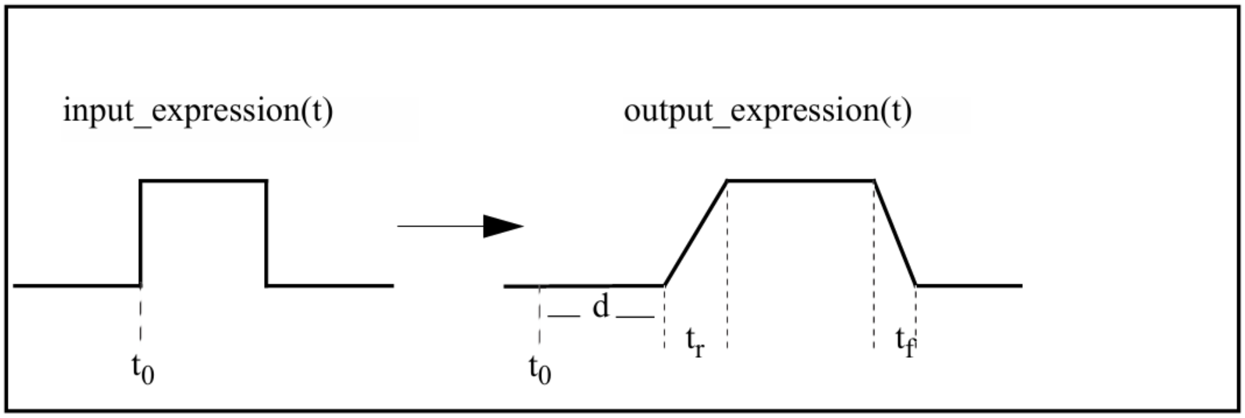


Figure 11 : Verilog-AMS transition filter

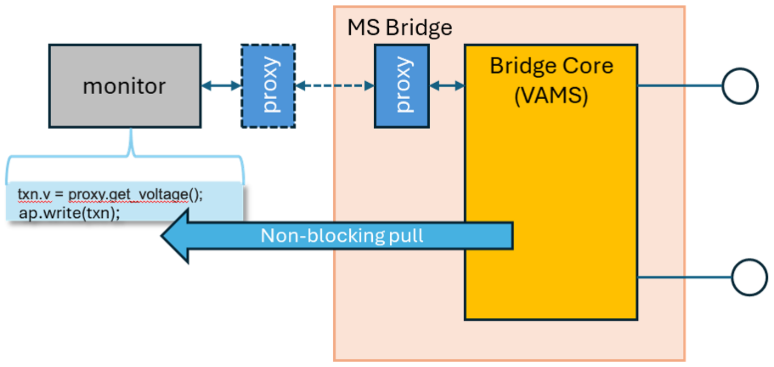
Within Verilog-AMS, there is no signal to say that the transition filter has completed, even though the simulator has that information. But Verilog-AMS code can detect whether the transition filter has completed, which is useful feedback to an agent. The input value to the transition filter can be compared to the output, and when they are within a certain tolerance, the filter can be deemed to have completed.

In the example below, the driver passes a voltage change to the Verilog-AMS bridge core via the proxy. The `set_ramp_voltage` function in the proxy calls the corresponding function in the bridge core. It waits for the bridge core function to complete, initializes the "end of transition" flag `vdc_eot`, and then returns to the driver. The driver waits for the flag to become 1.

```
// in the driver
   proxy.set_ramp_voltage(5.0);
   wait(proxy.vdc_eot);

module ms_bridge .....
   ms_bridge_core icore (...); // instantiation of the bridge_core

   class MSproxy extends vdriver_proxy; // vdriver_proxy defines int vdc_eot

      function automatic void set_ramp_voltage(real val, tr = 1e-9, tf = 1e-9);
         void'(i_core.set_ramp_voltage(val, tr, tf));
         vdc_eot = i_core.vdc_eot;
      endfunction: set_ramp_voltage
    ...
    endclass     MSproxy i_proxy = new("i_proxy");

    // Capture changes from the bridge core and copy to proxy.
    // @(..) at the end so that i_proxy.vdc_eot is assigned when the process starts.
    // This saves an assignment initialization statement.
    always begin
       i_proxy.vdc_eot = i_core.vdc_eot;
       @(i_core.vdc_eot);
    end
...
endmodule

// Verilog-AMS
module ms_bridge_core ....

   function automatic integer set_ramp_voltage(input real val, tr, tf);
      begin
         vdc     = val;
         vdc_tr  = tr;
         vdc_tf  = tf;
         vdc_eot = (abs(vdc_tran - vdc) < tol) ? 1:0; // Update vdc_eot
         set_ramp_voltage = 1; // Return 1; V-AMS does not have void functions
      end
   endfunction

   integer analog_clk;
   analog begin
        vdc_tran = transition(vdc, 0, vdc_tr, vdc_tf);
        // A change in vdc is a D2A even, triggering the analog solver.
        ...
        analog_clk = 1 - analog_clk; // Generate signal to update vdc_eot
   end

   // Update vdc_eot on every analog timestep.
   always@(absdelta(analog_clk,1,0,0)) vdc_eot  = (abs(vdc_tran - vdc) < tol) ? 1:0;
...
endmodule
```

In this example, `vdc` is a real value passed from the agent into the bridge core as a digital owned variable. `vdc_tran` is the output of the transition filter and is a continuous real variable. At each analog timestep, the output of the transition filter is updated and `vdc_eot` is recalculated. `vdc_eot` determines whether the continuous variable is within a tolerance of the discrete value. A tolerance is needed as the comparison is on two real numbers. In this case, the two real numbers represent voltages, so the recommendation for `tol` is the Verilog-AMS **`abstol`** for that discipline.

6.1.6 Pull analog value



```
txn.v = proxy.get_voltage();
ap.write(txn);
```

Figure 12 : Pulling a value from the bridge core

The pull pattern takes an instantaneous measurement from the bridge core such that it is suitable for a UVM monitor. It is recommended that the monitor call a function in the proxy that in turn calls a function in the bridge core. In Figure 12, `ap` is the instance name of a *uvm_analysis_port*.

6.2 DMS communication

DMS as defined in this document uses native SystemVerilog constructs, and could have been set up without a bridge and then the testbench and data communication would be simpler. However, in many cases, a DMS model either needs to use an AMS model as a baseline to fine-tune its behavior, or both models need to run interchangeably. These cases require setting up the DMS bridge core communication to be functionally identical to the AMS version and to inherit the Bridge Core-Proxy-Agent communication pattern described above. Based on this principle, the HDL and UVM testbench hierarchies could remain the same while switching between AMS and DMS modules at elaboration time.

This section describes how to implement in DMS the same communication patterns described in the previous section for AMS.

6.2.1 Event generation from analog signals

In DMS, digital events could be generated by monitoring value changes of variables. However, the user needs to guard against an excessive number of events due to noise. It is therefore important to utilize some tolerance. There are multiple ways to achieve this, e.g., instead of simply `@(myvar)`, it could be better to write "`@(abs(myvar - myvar_prev) - tol)`" on a sensitive signal, where **`abs()`** is a user-defined absolute value function.

Dedicated modules can also be created to achieve these kinds of monitoring. For simplicity, a delay could also be used for stimuli with predetermined duration (e.g., PWL or periodic waveforms) where end-of-signal transitions do not necessarily indicate the end of the transaction. One useful programming pattern is to set the timescale of the module to, e.g., one second, with a **`timeunit`** statement. A delay could then be sent from the UVM side in units of seconds, and the delay implemented as `#(delay_exp_in_sec * 1s)`.

6.2.2 Quantized continuously monitored analog signals

In DMS, analog signals can be directly monitored.

6.2.3 Event-driven queries of analog values

In order to preserve the function signatures and ensure a smooth transition between DMS and AMS, the proxy is used to query values from the bridge core. DMS implementations need to accommodate the same set of functions. The exact implementation of such functions may depend on the abstraction.

For example, when the analog output is modelled with the `VIZ_wire` user-defined nettype, the API could be defined as:

```
typedef struct {
  real  V;
  real  I;
  real  Z;
} VIZ_type;

nettype VIZ_type VIZ_wire;

module ms_bridge_core(inout VIZ_wire p);

  function real get_voltage(input dummy);
    begin
       get_voltage = p.V;
    end
  endfunction: get_voltage

  function real get_current(input dummy);
    begin
       get_current = p.I;
    end
  endfunction: get_current
endmodule
```

6.2.4 Push analog control

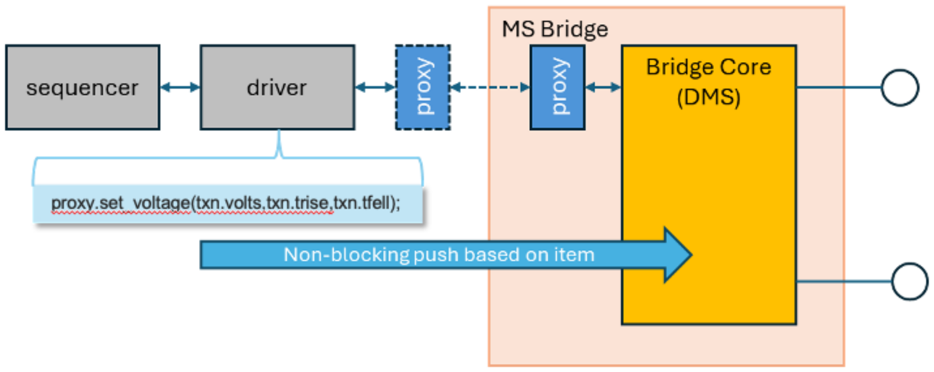

MS Bridge
sequencer
driver
proxy
proxy
Bridge Core
(DMS)
proxy.set_voltage(txn.volts,txn.trise,txn.tfell);
Non-blocking push based on item

Figure 13 : Push analog control – DMS version

As in AMS, the driver calls a method in the proxy class to make an instantaneous change to a bridge core value and then returns control to get the next sequence item. The `set_voltage()`/`get_voltage()` functions could remain mostly the same.

Similar to the transition filter in Verilog-AMS, in DMS one could create a dedicated function to control the updates to the voltage magnitude and frequency actually driven to the DUT and ensure that the changes are smooth. Instead of directly writing changes to the core module, the changes are intercepted and filtered by intermediate signals. This could improve performance by preventing too-frequent writes that would force the analog solver to reevaluate the analog matrix.

6.2.5 Push analog control with handshake (push-sync)

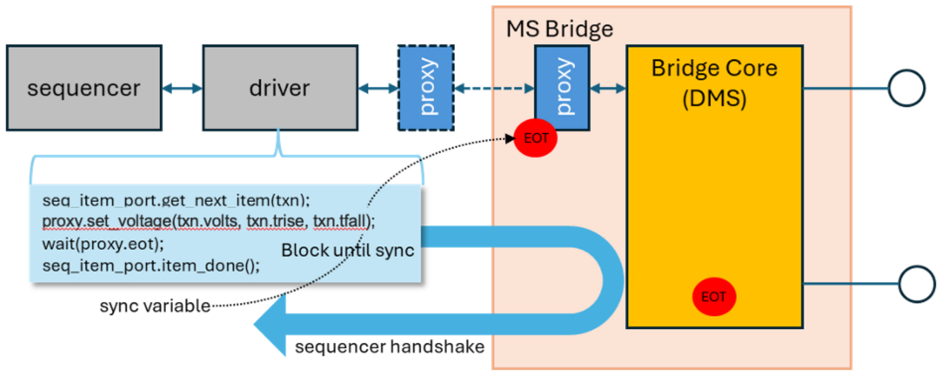
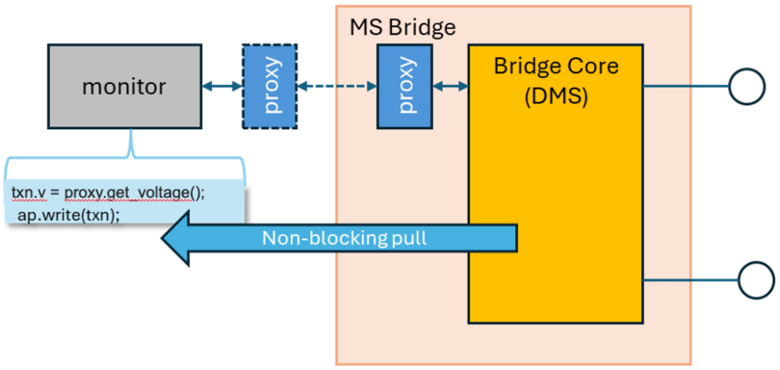


Figure 14 : Push-sync pattern for analog control example – DMS version

This push-sync pattern is similar to that of the AMS version. In the bridge core, the value update could be interpolated in a fashion similar to that of a transition filter.

6.2.6 Pull analog value



```
txn.v = proxy.get_voltage();
ap.write(txn);
```

Figure 15 : Pulling a value from the bridge core – DMS version

This pattern is also similar to that of AMS. In AMS, these queries need to be carried out by access functions, whereas in DMS, bridge core variables can be queried directly. User can also incorporate current or resistance with user defined net types into the model and enable queries of such signals.

7. Messaging

7.1 General

Users need the ability to report what is happening during simulation. Users may want to display:

- Errors, both in creating the verification environment and in simulation results
- Debug messages, to help understand why problems have occurred
- Progress reports, to show that the simulation is doing what it should

Some messages (e.g., errors) will be more important than others (e.g., progress reports). Therefore, users need the ability to categorize messages by severity:

- Fatal – e.g., the simulation has failed and cannot continue
- Error – e.g., there is a problem, but simulation can continue
- Warning – e.g., there is a possible problem or anticipated issue
- Info – e.g., for progress reports

Severity codes allow users to link specific actions with messages, for example, to stop the simulation on a fatal message or after a certain number of error messages. A summary of the messages of each severity at the end of the simulation enables the user to understand the simulation results at a glance. A standalone messaging mechanism for UVM is needed to allow separate control of simulation and simulator reports and to ensure portability, as shown in Figure 16.

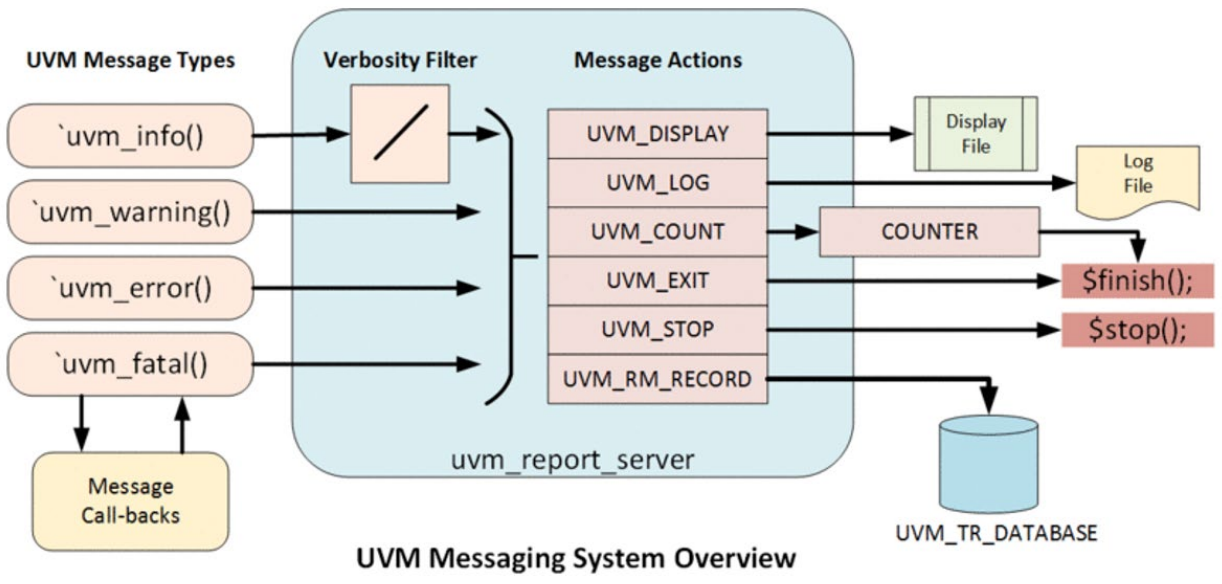


Figure 16 : UVM messaging system overview

7.2 UVM messaging from the bridge core

It is a requirement to implement UVM messaging from the bridge core. However, reporting macros are not supported in Verilog-AMS modules. A solution to this is via *upwards name referencing*.

For Verilog-AMS files, the **`include** file "**uvm_ms_includes.vamsh**" defined in this standard, found in Annex A, contains local parameters that define UVM verbosity levels as integers to match the UVM enums. The file "**uvm_ms_includes.svh**" included in the top-level SystemVerilog file of the MS Bridge declares void functions that wrap calls to the **uvm_report_*()** reporting functions.

Within a digital block of a Verilog-AMS file, users can call **`uvm_ms_[info|warning|error|fatal](…)** and upwards name referencing finds the corresponding function in the MS Bridge file. See Annex A for more details. SystemVerilog files in the bridge that need to call reporting functions can include either of the two files. The output messages report the file name and the line number from where the reporting function was called. An example is shown below in 7.2.1.

As in **`uvm_info**, the user specifies the verbosity in the call to **`uvm_ms_info**. If the verbosity argument is omitted (legal only in SystemVerilog), the default verbosity is **UVM_MEDIUM**. The verbosity level for **`uvm_ms_warning**, **`uvm_ms_error**, and **`uvm_ms_fatal** is **UVM_NONE**, as with their regular UVM counterparts.

7.2.1 UVM messaging from an AMS bridge core

Calling user-defined non-analog functions is not allowed in a Verilog-AMS `analog` block, but a solution to invoke messaging is to set a string value and toggle an integer, then use **absdelta()** to trigger on the toggle and read the string to call **`uvm_ms_*(…).**

Example:

```
parameter string P__TYPE = “vdriver”;
...
//Convert the detection in the analog block to a UVM report.
string message;
always@(absdelta(I_thr_triggered,1,0,0,1)) begin
```

```
   if(I_thr_triggered) begin
     $sformat(message,"The Current is above the threshold @ %eA",I_PLUS);
     `uvm_ms_info(P__TYPE,message,UVM_MEDIUM)
   end
 end
```

A typical output message might look like this:

```
UVM_INFO ../vdriver_bridge/vdriver_core.sv(377) @ 6400000000.00000 fs: reporter [vdriver] The
Current is above the threshold @ 1.000265e+00A
```

7.2.2 UVM messaging from a DMS bridge core

In principle, a digital core model could use regular UVM messaging, but it is recommended to use the UVM-MS messaging functions for consistency of reported contexts.

8. Testbench

Unlike classical UVM, UVM-MS needs to provide support for designs in a schematic capture tool that require some sort of netlist extraction. To facilitate this, a “dual-top” configuration is suggested. One of the top-level modules, called the *test_bench*, encapsulates the physical DUT and MS Bridge components. The other top-level module, called the *test_env*, encapsulates the UVM test, environment, agents, etc. This approach allows the *test_env* and associated collateral to be different for different netlist extractions of the *test_bench*. For example, depending on whether the *test_bench* is extracted as AMS or DMS, a different *test_env* could be used. In addition, the various schematic capture tools can deal with the complexities of creating design configurations to support instance view bindings.

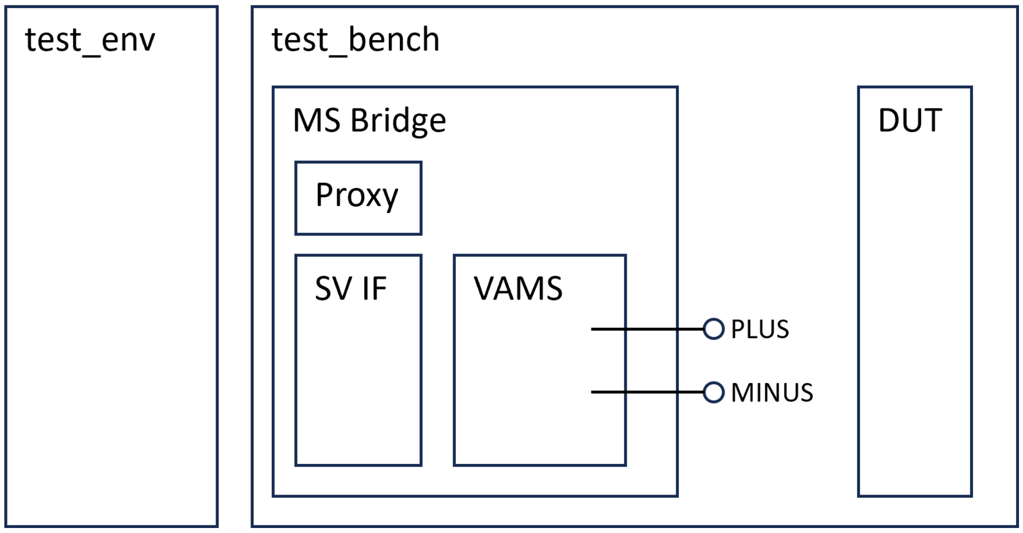


Figure 17 Dual-top UVM-MS hierarchy

9. (informative) Verilog-AMS circuit initialization

9.1 General

Understanding how a Verilog-AMS simulation is initialized can help avoid issues that cause time to debug. The Verilog-AMS standard defines the order, but a simplified description is presented here for convenience.

1. Variables are initialized, either to their default values or as specified in a variable declaration assignment. (In this, Verilog-AMS behaves like SystemVerilog, and not like Verilog, where variable declaration assignments execute at the same time as **`initial`** blocks.)

2. **`analog initial`** blocks execute. These blocks may not contain OOMRs, and the order of execution between **`analog initial`** blocks in different modules does not matter. These statements occur before the analog matrix is formed, so it is not possible here to probe a voltage, for example. These blocks are used for setting internal variables.

3. Digital **`initial`** and **`always`** blocks begin to execute in a nondeterministic order until they start to consume time. These blocks may contain OOMRs. So a digital **`initial`** block may query a value from an analog block. Thus, it is important to ensure that the values of analog variables read in this step are set correctly in step 2.

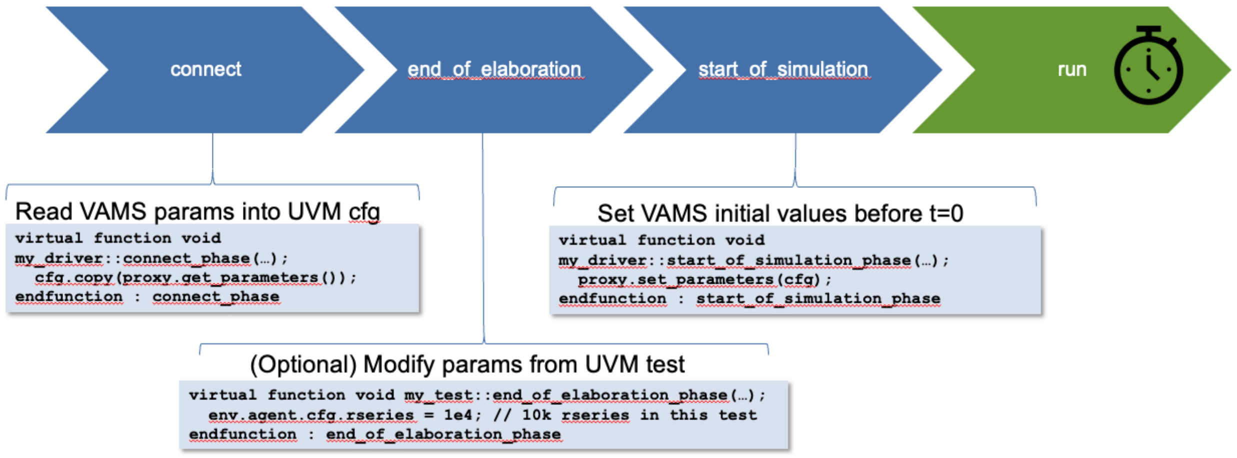
4. At this point, control passes over to the analog solver, which will try to find a DC operating point. Analog DC analysis and the digital simulation at time 0 execute iteratively until all signals at the A/D boundaries reach steady state, which is the DC operating point for transient and AC analysis.

9.2 UVM phases

Finding a DC operating point is an iterative process because it will try to converge. This may cause an analog-to-digital event to happen. The RTL code may do something that would then cause a digital-to-analog event that then changes the analog solution. This iteration might have to repeat a few times. The best thing to do is to make sure that the A-to-D or D-to-A boundaries are static. One way to do that is to start the supply zero and then ramp everything up. Assuming all of that works, a DC operating point will be obtained, and the analog solver will choose a point in time to move to and then pass back control to the digital simulator.

This provides an opportunity in the pre-run phases to reconfigure analog parameters. For example, it is possible to change a capacitor value or reconfigure resistors acting as switches to be open or closed before finding a DC operating point. This can be done in a pre-run phase after *build*.

For example, there is a capacitor and it has parameter values on it. In the *connect_phase*, the parameters can be read into the UVM configuration. The configuration could then be modified in the *end_of_elaboration* phase. Finally, in the *start_of_simulation* phase, the values on the capacitor can be updated to the values contained in the configuration. In the *run_phase*, no analog values should be changed until some time has been consumed by the digital simulator (e.g., `#1`). This ensures that the DC operating point of the circuit is retained.


connect
end_of_elaboration
start_of_simulation
run
Read VAMS params into UVM cfg
Set VAMS initial values before t=0
(Optional) Modify params from UVM test

```
virtual function void
my_driver::connect_phase(…);
  cfg.copy(proxy.get_parameters());
endfunction : connect_phase
```

```
virtual function void
my_driver::start_of_simulation_phase(…);
  proxy.set_parameters(cfg);
endfunction : start_of_simulation_phase
```

```
virtual function void my_test::end_of_elaboration_phase(…);
  env.agent.cfg.rseries = 1e4; // 10k rseries in this test
endfunction : end_of_elaboration_phase
```

Figure 18 : UVM pre-run phases

10. Known limitations

10.1 Inout ports on the SystemVerilog interface

Driver-receiver segregation is needed on `inout` connections from the SV IF to the bridge core when the DUT I/O abstraction is a user-defined type or electrical. The current workaround is to split the SV IF port into a driver and receiver manually.

10.2 Unpacked array ports on DUT

Using a DUT I/O bus of an unpacked array type, such as an array of real values, prevents swapping the abstraction of the bridge core between DMS and AMS. Verilog-AMS does not yet support unpacked array ports.

Annex A (normative) UVM-MS package and include files

The standard package **uvm_ms_pkg** contains the definition of the abstract base proxy class **uvm_ms_proxy**.

The *MS Bridge* and other SystemVerilog files should include the "**uvm_ms_includes.svh**” file in addition to the standard UVM package and include file(s). Verilog-AMS modules in the MS Bridge hierarchy need to include “**uvm_ms_vamsh**”.

A.1 uvm_ms_pkg.sv

```
// ===================================================
// Proxy class to be extended in the MS Bridge module.
// It allows UVM components to access proxy API by
// passing a handle to the proxy instance via uvm_config_db.
// Class is declared virtual so it cannot
// be instantiated, it must be derived.
// ===================================================

`ifndef UVM_MS_PKG_SV
`define UVM_MS_PKG_SV

`include "uvm_macros.svh"

package uvm_ms_pkg;
  import uvm_pkg::*;

  /* uvm_ms_proxy provides a communication API between driver/monitor and the bridge.
   * A proxy specific to the bridge should be created inside the bridge by extending
   * uvm_ms_proxy and then implementing the API functions accordingly.
   */

  virtual class uvm_ms_proxy;
    string name;

    function new(string name="uvm_ms_proxy");
      this.name = name;
    endfunction

    // Example prototype for push/pull functions to core
    // This function should be added to a derived class and NOT to the uvm_ms_proxy class
    // virtual function void push(input real ampl, bias, freq, bit enable);
    //   `uvm_warning("proxy","Function push not implemented")
    // endfunction
    // virtual function void pull(output real ampl, bias, freq, bit enable);
    //   `uvm_warning("proxy","Function pull not implemented")
    // endfunction

  endclass : uvm_ms_proxy

endpackage : uvm_ms_pkg

`endif
```

A.2 uvm_ms_includes.svh

```
// ======================================================================
// Must be `included within a module scope
// Wrapper functions to hook the bridge core digital calls to the UVM
// reporting system. Upwards name referencing is used so it works in
// various abstractions of the bridge core, e.g., both SV and Verilog-AMS.
//
// The UVM messaging macros are based on class hierarchy. UVM-MS uses different
// macro names to distinguish them from the UVM macros.
//
// `uvm_ms_info works the same as `uvm_info except that it passes `__FILE__ and
// `__LINE__ arguments pass to the info message, reporting where the macro is called,
// typically the bridge core
//
// Output format:
// UVM_INFO <file path>(<line number>) @ <time>: reporter <tag> <message>
// ("reporter" is the typical default UVM report handler name
// ======================================================================
`define uvm_ms_info(id,message,uvm_verbosity) \
                        uvm_ms_info(id,message,uvm_verbosity,`__FILE__ ,`__LINE__);

`define uvm_ms_warning(id,message) uvm_ms_warning(id,message,`__FILE__ ,`__LINE__);

`define uvm_ms_error  (id,message) uvm_ms_error  (id,message,`__FILE__ ,`__LINE__);

`define uvm_ms_fatal  (id,message) uvm_ms_fatal  (id,message,`__FILE__ ,`__LINE__);


// The uvm_ms_info function takes arguments from the `uvm_ms_info macro and recreates
// a message identical to `uvm_info. This is needed because `uvm_ms_info is being
// called from a module representing the bridge core and not from a UVM component.

function void uvm_ms_info(string id, string message, int verbosity, string file,
                                                                       int line);
  uvm_report_info(id,message,verbosity,file,line);
endfunction: uvm_ms_info

function void uvm_ms_warning(string id, string message, string file, int line);
  uvm_report_warning(id, message,, file, line);
endfunction: uvm_ms_warning

function void uvm_ms_error(string id, string message, string file, int line);
  uvm_report_error(id, message,, file, line);
endfunction: uvm_ms_error

function void uvm_ms_fatal(string id, string message, string file, int line);
  uvm_report_fatal(id, message,, file, line);
endfunction: uvm_ms_fatal
```

A.3 uvm_ms_includes.vamsh

```
// `include file for Verilog-AMS. Not needed for SystemVerilog.
// enums and dynamic objects cannot be accessed via OOMRs,
// so local parameters are used.
// Must be `included within a module scope

localparam integer UVM_NONE   = 0;
localparam integer UVM_LOW    = 100;
localparam integer UVM_MEDIUM = 200;
localparam integer UVM_HIGH   = 300;
localparam integer UVM_FULL   = 400;
localparam integer UVM_DEBUG  = 500;

// UVM messaging can be called from Verilog-AMS digital code using these functions.
// Via upwards name referencing, the code calls the function in the SystemVerilog
// wrapper layer that then calls the UVM macros (see file "uvm_ms_includes.svh").
// The system can also be used in SystemVerilog implementations of the bridge core.

`define uvm_ms_info(id,message,verbosity) \
                            uvm_ms_info(id,message,verbosity,`__FILE__ ,`__LINE__);

`define uvm_ms_warning(id,message) uvm_ms_warning(id,message,`__FILE__ ,`__LINE__);

`define uvm_ms_error  (id,message) uvm_ms_error  (id,message,`__FILE__ ,`__LINE__);

`define uvm_ms_fatal  (id,message) uvm_ms_fatal  (id,message,`__FILE__ ,`__LINE__);
```

Annex B
(informative) Example: migrating UVM to UVM-MS

B.1 Overview

The example presented here is an oscillator agent used to drive and monitor a clock signal to and from a frequency adapter DUT. A block diagram of the DUT is shown in Figure 19:

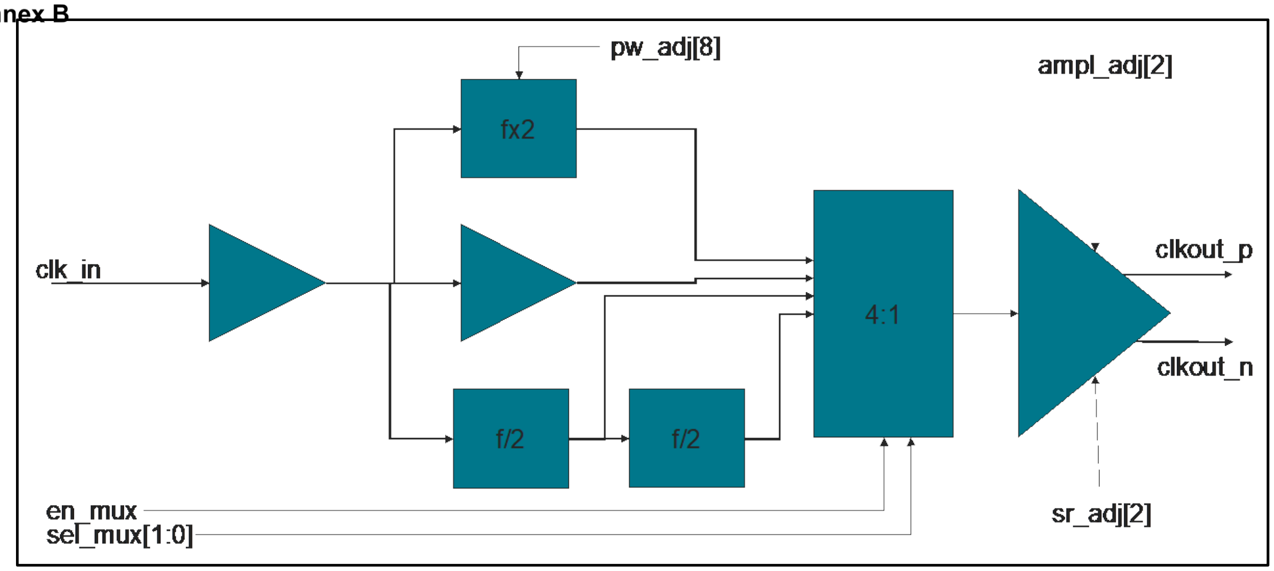


Figure 19 : Frequency adapter DUT

The frequency adapter DUT has a single-ended clock input and a differential clock output. Five control signals modify the clock input to produce the clock outputs:

- `en_mux` enables the multiplexer.
- `sel_mux` selects one of four multiplication factors – 2X, 1X, 0.5X, 0.25X.
- `pw_adj` is an 8-bit signal that adjusts the pulse width of the 2X clock.
- `ampl_adj` is a 2-bit signal that adjusts the amplitude of the output clocks.
- `sr_adj` is a 2-bit signal that adjusts the slew rate of the output clocks (only applicable for DMS/AMS model).

There are two oscillator agents, one for generating the input clock and one for detecting the output clock, and an I2C agent for programming the registers with the values used to drive the control signals.

The UVM testbench, including two instances of the oscillator interface **osc_if**, is shown in Figure 20:

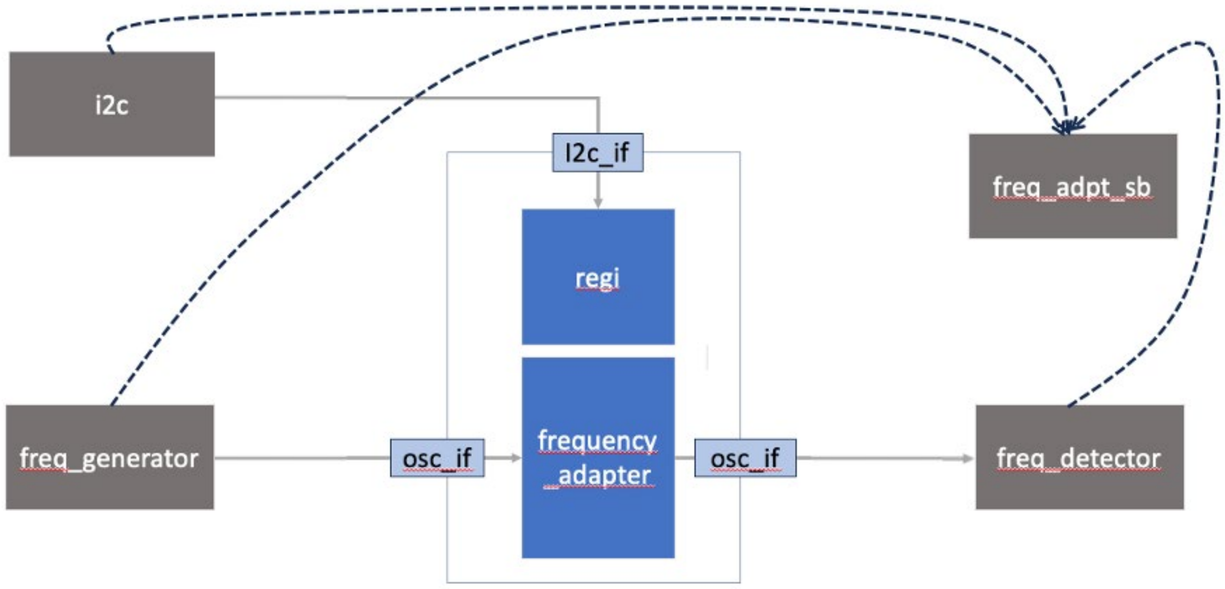


Figure 20: UVM testbench, showing oscillator interface

The frequency adapter DUT can be implemented as a pure digital (SystemVerilog), DMS (SystemVerilog/Verilog-AMS) or AMS (Verilog-AMS/SPICE) model, with the **clk_in**, **clkout_p**, and **clkout_n** ports implemented as pure digital, DMS or AMS signals respectively. Code snippets showing the respective module declarations are shown below:

```
// digital
module frequency_adapter (
  output logic clkout_p, clkout_n,                     // differential output
  input  logic clk_in,                                 // clock input
  input  logic en_mux, [1:0] sel_mux,                  // register control
  input  logic [7:0] pw_adj, [1:0] sr_adj, ampl_adj);  // digital control voltage

// DMS
module frequency_adapter import rnm_pkg::*; (
  output wreal4state clkout_p, clkout_n,               // differential output
  input  wreal4state clk_in,                           // clock input
  input  logic en_mux, [1:0] sel_mux,                  // register control
  input  logic [7:0] pw_adj, [1:0] sr_adj, ampl_adj);  // digital control voltage

// AMS
module frequency_adapter(clkout_p, clkout_n, clk_in, en_mux, sel_mux, pw_adj, sr_adj, ampl_adj);
  output clkout_p, clkout_n;                           // differential output
  input clk_in;                                        // clock input
  input en_mux, [1:0] sel_mux;                         // register control
  input [7:0] pw_adj, [1:0] sr_adj, ampl_adj;          // digital control voltage
  electrical clkout_p, clkout_n, clk_in;
```

This example assumes that pure digital versions of the frequency adapter DUT and the oscillator agent existed previously. Therefore, the new DMS and AMS DUTs necessitate the creation of DMS and AMS versions of the oscillator agent.

The full working testcase is available in the **frequency_adapter** directory of the UVM-MS Working Group on the Accellera website: https://accellera.org/downloads/drafts-review.

B.2 Creating the bridge module

The original interface for the oscillator (**`osc_if`**) communicates with the DUT through the **`osc_clk`** net for a single-ended clock, or **`osc_clk_p`** and **`osc_clk_n`** for a differential clock, as shown in the snippet below:

```
interface osc_if ();
...
  // DUT signals
  wire osc_clk = clk;          // Single-ended clock
  wire osc_clk_p, osc_clk_n;  // Differential clock
...
endinterface
```

The interface is instantiated twice. **`generator_if`** connects the agent driver and the DUT input, and **`detector_if`** connects the agent monitor and the DUT outputs.

For an MS environment, the MS Bridge that is created has to append to or replace this interface in function. Therefore, the ports of the MS Bridge (**`osc_bridge`**) will match the above nets. The bridge module will also contain the instantiation of the bridge core (**`osc_bridge_core`**) and the implementation of the bridge proxy (**`osc_bridge_proxy`**, which extends the base class **`osc_proxy`**). The bridge module is also instantiated twice, like the interface, once for the generator and once for the detector.

```
module osc_bridge (
  output interconnect osc_clk,
  input  interconnect osc_clk_p,
  input  interconnect osc_clk_n
  );

  // UVM + MS extras
  import uvm_pkg::*;
  import uvm_ms_pkg::*;
  `include "uvm_macros.svh"
  `include "uvm_ms_includes.svh"

  // UVM package for this component
  import osc_pkg::*;

  // The clock detector can be configured in active or passive mode using the "passive" parameter,
  // and to choose between single-ended or differential clock using the "diff_sel" parameter
  parameter bit diff_sel = 0;
  parameter bit passive = 0;

  // Class osc_bridge_proxy extends the osc_proxy included in osc_pkg.sv
  class osc_bridge_proxy extends osc_bridge_proxy;
    function new(string name = "");
      super.new(name);
    endfunction : new

    // implementation of pull function from base class not needed
    // implementation of push function from base class
    function void push(input real ampl, bias, freq, bit enable);
      core.ampl_in  = ampl;
      core.bias_in  = bias;
      core.freq_in  = freq;
      core.enable   = enable;
    endfunction

  endclass

  osc_bridge_proxy proxy = new("proxy");

...

  // Bridge core instantiation
  osc_bridge_core #(.diff_sel(diff_sel), .passive(passive)) core (
    .osc_clk  (osc_clk),
    .osc_clk_p(osc_clk_p),
    .osc_clk_n(osc_clk_n)
    );

endmodule
```

B.3 Extending the driver and monitor classes

The mixed-signal version of the oscillator driver class (**`osc_ms_driver`**) uses the bridge proxy to read and write from the bridge core. The mixed-signal driver extends the digital driver (**`osc_driver`**), so the interface declaration and connection in the base class are inherited by the extended class. Code snippets are shown below Key items to note:

1. The mixed-signal driver **`osc_ms_driver`** extends the digital driver **`osc_driver`**.

2. The mixed-signal driver instantiates a variable of base proxy type (**`osc_proxy`**) and assigns the variable a handle to the bridge proxy in a way similar to how the virtual interface **`vif`** is assigned a handle to the **`osc_if`** interface in the digital driver, with a call to **`uvm_config_db::get`**`()` in the UVM *connect* phase.

3. The bridge core is driven and monitored using calls to bridge proxy methods and references to variables in the proxy.

```
//---------------------------
// CLASS: osc_driver
//---------------------------
class osc_driver extends uvm_driver #(osc_transaction);

  // The virtual interface used to drive and view HDL signals.
  virtual interface osc_if vif;

  // Count transactions sent
  int num_sent;

  // period of the generated clock
  real period;

        // component macro
  `uvm_component_utils_begin(osc_driver)
    `uvm_field_int(num_sent, UVM_ALL_ON)
    `uvm_field_real(period, UVM_ALL_ON)
  `uvm_component_utils_end

  int test_config;

  // Constructor - required syntax for UVM automation and utilities
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
  endfunction

  function void connect_phase(uvm_phase phase);
    if (!uvm_config_db#(virtual osc_if)::get(this,"","vif", vif))
      `uvm_error("NOVIF",{"virtual interface must be set for: ",get_full_name(),".vif"})
  endfunction: connect_phase
  ...

endclass : osc_driver

//---------------------------
// CLASS: osc_ms_driver
//---------------------------
class osc_ms_driver extends osc_driver;

  protected osc_proxy bridge_proxy;

  osc_ms_transaction ms_req;

  // Get value to drive onto diff_sel
  bit diff_sel;

  // Provide implmentations of virtual methods such as get_type_name and create
  `uvm_component_utils_begin(osc_ms_driver)
    `uvm_field_int(diff_sel, UVM_ALL_ON)
  `uvm_component_utils_end

  // Constructor - required syntax for UVM automation and utilities
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new

...
```

```
  function void build_phase(uvm_phase phase);
        super.build_phase(phase);
    //TODO: Put this in the connect_phase()
    if(!uvm_config_db#(osc_proxy)::get(this,"","bridge_proxy",bridge_proxy))
      `uvm_error(get_type_name(),"bridge proxy not configured");
  endfunction

endclass : osc_ms_driver

// UVM run_phase()
task osc_ms_driver::run_phase(uvm_phase phase);
  super.run_phase(phase);
  get_and_drive();
endtask

// Gets transactions from the sequencer and passes them to the driver.
task osc_ms_driver::get_and_drive();
  forever begin
    // Get new item from the sequencer
    seq_item_port.get_next_item(req);
    $cast(ms_req,req); //Optionally add check on return value of $cast()
    // Drive the item
    drive_transaction(ms_req);
    fork
      #(200*1ns); //Time for transaction
      begin : sample_thread
        #(1ns) bridge_proxy.sampling_do = 1;
        #(1ns) bridge_proxy.sampling_do = 0;
      end
    join
    // Communicate item done to the sequencer
    seq_item_port.item_done();
  end
endtask : get_and_drive
```

The **`osc_monitor`** class is similarly extended to **`osc_ms_monitor`**, as shown below.

```
//---------------------------
// CLASS: osc_monitor
//---------------------------
class osc_monitor extends uvm_monitor;

  // Virtual Interface for monitoring DUT signals
  virtual interface osc_if vif;
...
  osc_transaction osc_clk_transaction, osc_clk_p_transaction;

  // This TLM port is used to connect the monitor to the scoreboard
  uvm_analysis_port #(osc_transaction) item_collected_port;
...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    if (!uvm_config_db#(virtual osc_if)::get(this, get_full_name(), "vif", vif))
      `uvm_fatal("NOVIF",{"virtual interface must be set for: ",get_full_name(),".vif"})
    if (!uvm_config_int::get(this,"","diff_sel", diff_sel))
      `uvm_fatal("NOCONFIG",{"value must be set for: ",get_full_name(),".diff_sel"})
    else      `uvm_info("CONFIG_CORRECT",{"Value   of   ",get_full_name(),   $sformatf(".diff_sel
= %d",diff_sel)}, UVM_MEDIUM)
  endfunction: connect_phase
...
endclass : osc_monitor

//---------------------------
// CLASS: osc_ms_monitor
//---------------------------
class osc_ms_monitor extends osc_monitor;

  // Virtual interface for monitoring DUT signals
  protected osc_proxy bridge_proxy;

  // Count transactions collected
  int num_col;
...
  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
      if(!uvm_config_db#(osc_proxy)::get(this,"","bridge_proxy",bridge_proxy))
        `uvm_error(get_type_name(),"bridge proxy not configured");
  endfunction
...
endclass : osc_ms_monitor
...
task osc_ms_monitor::collect_transaction();
    // This monitor re-uses its data items for ALL transactions
    transaction = osc_ms_transaction::type_id::create("transaction", this);
    forever begin
      @(posedge bridge_proxy.sampling_done);
      // Begin transaction recording
      void'(begin_tr(transaction, "analog_clock source monitor"));
      transaction.data_type = OSC_MS_SAMPLE;
      transaction.ampl = bridge_proxy.ampl_out;
      transaction.bias = bridge_proxy.bias_out;
      transaction.freq = bridge_proxy.freq_out;
      `uvm_info(get_type_name(),
        $sformatf("source transaction collected :\n%s", transaction.sprint()), UVM_HIGH)
                                                               //Temporarily dropped verbosity
      if (checks_enable)   perform_checks();
      if (coverage_enable) perform_coverage();

      // Send transaction to scoreboard via TLM write()
//UPDATE - Make copy of transaction and write the copy
      item_collected_port.write(transaction);
      num_col++;

      fork
        begin : wait_for_sampling_done
          @(negedge bridge_proxy.sampling_done);
          disable wait_for_timeout;
        end
        begin : wait_for_timeout
          #150ns;
          disable wait_for_sampling_done;
        end
      join
      // End transaction recording
      end_tr(transaction);
    end
  endtask : collect_transaction
```

B.4 Extending the sequence item class

The mixed-signal agent uses the frequency, amplitude, and DC bias to generate the clock signal. These fields are also used to collect the information received by the monitor. The enable field is used to turn the clock generation on or off, and the delay and duration fields are used to set the delay before sampling and specify the number of cycles sampled.

The fields other than frequency (included in the digital sequence item, **`osc_transaction`**) are included in the mixed-signal agent's sequence item (**`osc_ms_transaction`**). **`osc_ms_transaction`** extends **`osc_transaction`** and retains the fields and constraints from the base class, since the extended class does not override them.

```
//----------------------------
// CLASS: osc_transaction
//----------------------------
class osc_transaction extends uvm_sequence_item;

  rand real freq; // frequency of input clock
  bit diff_sel;

  `uvm_object_utils_begin(osc_transaction)
    `uvm_field_real(freq, UVM_ALL_ON)
  `uvm_object_utils_end

  // Constraints go here
  soft constraint default_freq_c {
    freq > 5e8;
    freq < 1e9;
  }

  // Constructor - required syntax for UVM automation and utilities
  function new (string name = "osc_transaction");
    super.new(name);
  endfunction : new

endclass : osc_transaction

//----------------------------
// CLASS: osc_ms_transaction
//----------------------------
class osc_ms_transaction extends osc_transaction;

  rand osc_ms_data_type_e data_type;

  // Drive fields
  rand real ampl;
  rand real bias;
  rand bit enable;

  //Measurement fields
  rand real delay;    //Delay in ns
  rand int  duration;

  `uvm_object_utils_begin(osc_ms_transaction)
    `uvm_field_enum(osc_ms_data_type_e, data_type, UVM_DEFAULT)
    `uvm_field_real(ampl, UVM_DEFAULT)
    `uvm_field_real(bias, UVM_DEFAULT)
    `uvm_field_int(enable, UVM_DEFAULT)
    `uvm_field_real(delay, UVM_DEFAULT)
    `uvm_field_int(duration, UVM_DEFAULT)
  `uvm_object_utils_end

  // Constraints go here
  // To override, use the same constraint name or TCL to disable
  constraint default_drive_trans_c {
    ampl > 0.95;
    ampl < 1.65;
    bias inside {[-0.05:0.5]};
    enable dist {1'b0 := 1, 1'b1 := 5};
  }
  constraint default_measurement_trans_c {
    duration > 20;
    duration < 32;
    delay > 0.0;
    delay < 1.0;
  }

  // Constructor - required syntax for UVM automation and utilities
  function new (string name = "unnamed-osc_ms_transaction");
    super.new(name);
  endfunction : new

endclass : osc_ms_transaction
```

B.5 Extending the scoreboard class

The scoreboard has a **`write_osc_gen()`** function to record the attributes of the input clock and a **`write_osc_det()`** function to record the attributes detected on the output clocks. The **`write_i2c()`** function will record the register values driving the frequency adapter control signals. The scoreboard will use the values of the control registers to determine whether the output clock was generated correctly.

Since the monitors for the oscillator agent are extended to mixed-signal, including the sequence item that they operate on, the implementations of the scoreboard write functions need to be likewise extended. Key points:

1. The mixed-signal **`freq_adpt_ms_scoreboard`** class is extended from the digital **`freq_adpt_scoreboard`** class.

2. The **`write_osc_gen()`** and **`write_osc_det()`** methods are *overridden* in the mixed-signal scoreboard.

3. The **`write_i2c()`** method, which is the write function for the digital-only register monitor, is also overridden in the mixed-signal scoreboard.

B.6 Changes to testbench class

In order to incorporate the extended driver, monitor, sequence item and scoreboard classes, and handles to the new bridge proxies, the mixed-signal testbench class **`freq_adpt_ms_tb`** extends the digital-only testbench class **`freq_adpt_tb`**, with the following changes in the UVM *build* phase:

1. Set up pointer references for the bridge proxy in the generator and detector agents, using the **`uvm_config_db::set()`** method.

2. Override the driver, monitor, sequence item and scoreboard with the mixed-signal versions, using the **`set_type_override_by_type()`** method.

3. Call the base class's build phase with **`super.`**`build_phase()`.

```
// Scoreboard of the frequency adapter UVCs (freq_generator, freq_detector and registers)
// There will always be error messages to display the comparision results between
// expected output frequency (frequency_generator) and actual frequency (frequency_detector).

`uvm_analysis_imp_decl(_i2c)
`uvm_analysis_imp_decl(_osc_gen)
`uvm_analysis_imp_decl(_osc_det)

class freq_adpt_scoreboard extends uvm_scoreboard;

  typedef enum bit {COV_ENABLE, COV_DISABLE} cover_e;
  cover_e coverage_control = COV_ENABLE;

  // component utils macro
  `uvm_component_utils_begin(freq_adpt_scoreboard)
    `uvm_field_enum(cover_e, coverage_control, UVM_ALL_ON)
  `uvm_component_utils_end

  // define TLM port implementation object

  uvm_analysis_imp_i2c     #(i2c_packet     , freq_adpt_scoreboard) sb_i2c_in;
  uvm_analysis_imp_osc_gen #(osc_transaction, freq_adpt_scoreboard) sb_osc_gen;
  uvm_analysis_imp_osc_det #(osc_transaction, freq_adpt_scoreboard) sb_osc_det;
...
  // write() function for I2C registers block
  virtual function void write_i2c(i2c_packet packet);
```

```
    i2c_packet sb_packet;

    //Make a copy for storing in the scoreboard
    $cast(sb_packet, packet.clone()); // Clone returns uvm_object type
    i2c_packets_in++;

    if (sb_packet.tg_addr != sb_packet.tg_id || sb_packet.reg_addr > 2)
      i2c_in_drop++;
    else begin
      if(sb_packet.rw_ == 0)
        INT_REG[sb_packet.reg_addr] = sb_packet.data;
      MUX_REG = INT_REG[1];
      en_mux  = MUX_REG[2];
      sel_mux = MUX_REG[1:0];
    end
  endfunction

  // write() function for generator
  virtual function void write_osc_gen(osc_transaction packet);
    osc_transaction sb_packet;

    $cast(sb_packet, packet.clone());  // Clone returns uvm_object type
    freq_generator_packets_in++;
    freq_in_reg = sb_packet.freq;
  endfunction: write_osc_gen
// write() function for detector
  virtual function void write_osc_det(osc_transaction packet);
    osc_transaction sb_packet;
    $cast(sb_packet, packet.clone());  // Clone returns uvm_object type
    if(en_mux) begin
        // Compare output freq with expected result calculated from input freq
        actualresult_checker_p = sb_packet.freq;
...
    end
  endfunction: write_osc_det
...
endclass: freq_adpt_scoreboard


//---------------------------------------------
// Mixed-signal version of scoreboard
//---------------------------------------------

class freq_adpt_ms_scoreboard extends freq_adpt_scoreboard;
  `uvm_component_utils(freq_adpt_ms_scoreboard)

// variables of freq_generator
  real ampl, bias, freq;

  // variables for freq_detector coverage check and compare
  real freq_out_p, freq_out_n;
  real freq_tol;
  bit pktcompare;
...
  // write() function for I2C registers block
  virtual function void write_i2c(i2c_packet packet);
...
  endfunction

  // write() function for frequency generator
  virtual function void write_osc_gen(osc_transaction packet);
    osc_ms_transaction sb_packet;

    $cast(sb_packet, packet.clone());  // Clone returns uvm_object type

    freq_generator_packets_in++;
    freq = sb_packet.freq;
    ampl = sb_packet.ampl;
    bias = sb_packet.bias;

    `uvm_info("WRITE_OSC_GEN",
      $sformatf("\nFreq = %f\tAmpl = %f \t Bias = %f",freq,ampl,bias),UVM_LOW)

    osc_gen_change_observed = 1;
  endfunction: write_osc_gen

  // write() function for detector
  virtual function void write_osc_det(osc_transaction packet);
    osc_ms_transaction sb_packet;
    $cast(sb_packet, packet.clone());  // Clone returns uvm_object type
    freq_detector_packets_in++;
    // Compare immediately if the report from the detector was due to a change from the generator
    if(osc_gen_change_observed) begin
      if(en_mux) begin
        // Compare output freq with expected result calculated from input freq
  ...
  endfunction: write_osc_det
...
endclass: freq_adpt_ms_scoreboard
```

```
// tb class for digital version of frequency adapter
class freq_adpt_tb extends uvm_env;

  `uvm_component_utils(freq_adpt_tb)

  i2c_env i2c;
  osc_env freq_generator;
  osc_env freq_detector;

  freq_adpt_scoreboard freq_adpt_sb;

  function new (string name, uvm_component parent=null);
    super.new(name, parent);
  endfunction : new

  function void build_phase(uvm_phase phase);
    `uvm_info("MSG","In the build phase",UVM_MEDIUM)

    uvm_config_db#(virtual osc_if)::set(this,"freq_generator*", "vif", top.generator_if);
    uvm_config_db#(virtual osc_if)::set(this,"freq_detector*" , "vif", top.detector_if);
    uvm_config_db#(virtual i2c_if)::set(this,"i2c.agent.*"    , "vif", top.i2c_if);

    // config the value of diff_sel for freq_generator to 0 - single-ended clock generation
    // config the value of diff_sel for freq_detector  to 1 - differential clock detection
    uvm_config_int::set(this,"freq_generator.agent.*","diff_sel", 0);
    uvm_config_int::set(this,"freq_detector.agent.*" ,"diff_sel", 1);

    super.build_phase(phase);

    // create the envs for the generator, detector, registers and scoreboard
    freq_generator  = osc_env::type_id::create("freq_generator", this);
    freq_detector   = osc_env::type_id::create("freq_detector",  this);
    i2c             = i2c_env::type_id::create("i2c", this);
    freq_adpt_sb    = freq_adpt_scoreboard::type_id::create("freq_adpt_sb", this);
  endfunction : build_phase

  function void connect_phase(uvm_phase phase);
    // Connect the TLM ports from the UVCs to the scoreboard
    i2c.agent.monitor.item_collected_port.connect(freq_adpt_sb.sb_i2c_in);
    freq_generator.agent.monitor.item_collected_port.connect(freq_adpt_sb.sb_osc_gen);
    freq_detector.agent.monitor.item_collected_port.connect (freq_adpt_sb.sb_osc_det);
  endfunction : connect_phase
endclass : freq_adpt_tb
```

```
// tb class for SV-RNM and VAMS version of frequency adapter
class freq_adpt_ms_tb extends freq_adpt_tb;
```

```
  // component macro
  `uvm_component_utils(freq_adpt_ms_tb)

  // Constructor
  function new (string name, uvm_component parent=null);
    super.new(name, parent);
  endfunction : new

  // UVM build() phase
  function void build_phase(uvm_phase phase);
    `ifdef UVM_AMS
    // set up bridge proxy pointer references in generator and detector agents
    uvm_config_db #(osc_proxy)::set(this,"freq_generator.agent.*","bridge_proxy",
                                                top.generator_bridge.proxy);
    uvm_config_db #(osc_proxy)::set(this,"freq_detector.agent.*","bridge_proxy",
                                                top.detector_bridge.proxy);
    `endif

    // override driver, monitor, and scoreboard with UVM-AMS versions
    set_type_override_by_type(osc_transaction::get_type(),osc_ms_transaction::get_type());
    set_type_override_by_type(osc_driver::get_type()     ,osc_ms_driver::get_type());
    set_type_override_by_type(osc_monitor::get_type()    ,osc_ms_monitor::get_type());
    set_type_override_by_type(freq_adpt_scoreboard::get_type(),
                         (freq_adpt_ms_scoreboard::get_type());

    `ifdef DMS_I2C
    uvm_config_db #(i2c_proxy)::set(this,"i2c.agent.*","bridge_proxy", top.i2c_bridge.proxy);
    set_type_override_by_type(i2c_driver::get_type(),i2c_ms_driver::get_type());
    `endif

    super.build_phase(phase);

  endfunction
endclass : freq_adpt_ms_tb
```

B.7 Changes to top-level module

The only changes required for the top-level module are:

1. Importing the UVM MS package and including the UVM MS macros file.
2. Instantiating the MS bridges for the oscillator agents
3. Replacing the **`assign`** statements for the DUT's `clkin` and `clkout_p/n` ports with port connections to the respective MS bridges

```
module top;

  // Import the UVM libraries
  import uvm_pkg::*;
  import uvm_ms_pkg::*;

  // Include the UVM macros
  `include "uvm_macros.svh"
  `include "uvm_ms_includes.svh"

  // Import the UVC packages
  import osc_pkg::*;
  import i2c_pkg::*;

  // Include the test library files
  `include "freq_adpt_scoreboard.sv"
  `include "freq_adpt_tb.sv"
  `include "test_lib.sv"
...
  // Clock and reset signals
  wire clkout_p, clkout_n;
  wire clk_in;
...
  bit en_mux;
  bit [1:0] sel_mux;
  bit [1:0] ampl_adj;
  bit [1:0] sr_adj;
  bit [7:0] pw_adj;
...
  // Interfaces to the DUT
  osc_if generator_if ();
  osc_if detector_if ();
...
  // For a pure digital DUT, we map the inputs and outputs directly to the
  // generator and detector interface nets, respectively.
  // For a mixed-signal DUT (AMS/DMS), we instantiaTE MS bridges for the generator and detector UVCs.
  // These in turn instantiate bridge cores that perform the generation/detection operations for the UVCs.
  `ifdef UVM_AMS
  osc_bridge #(.diff_sel(0), .passive(0)) generator_bridge (.osc_clk(clk_in), .osc_clk_p(), .osc_clk_n());
  osc_bridge #(.diff_sel(1), .passive(1)) detector_bridge  (.osc_clk()      , .osc_clk_p(clkout_p),
                                                                              .osc_clk_n(clkout_n));
  `else
  assign clk_in = generator_if.osc_clk;
  assign detector_if.osc_clk_p = clkout_p;
  assign detector_if.osc_clk_n = clkout_n;
  `endif

  //Frequency Adapter DUT
  frequency_adapter #(.vsup(1.0)) DUT(
    .clk_in   (clk_in),
    .clkout_p (clkout_p),
    .clkout_n (clkout_n),
    .en_mux   (en_mux),
    .pw_adj   (pw_adj),
    .sel_mux  (sel_mux),
    .ampl_adj (ampl_adj),
    .sr_adj   (sr_adj)
  );
...
endmodule : top
```

The UVM-MS testbench, including two instances of the oscillator interface **`osc_if`**, is shown in Figure 21:

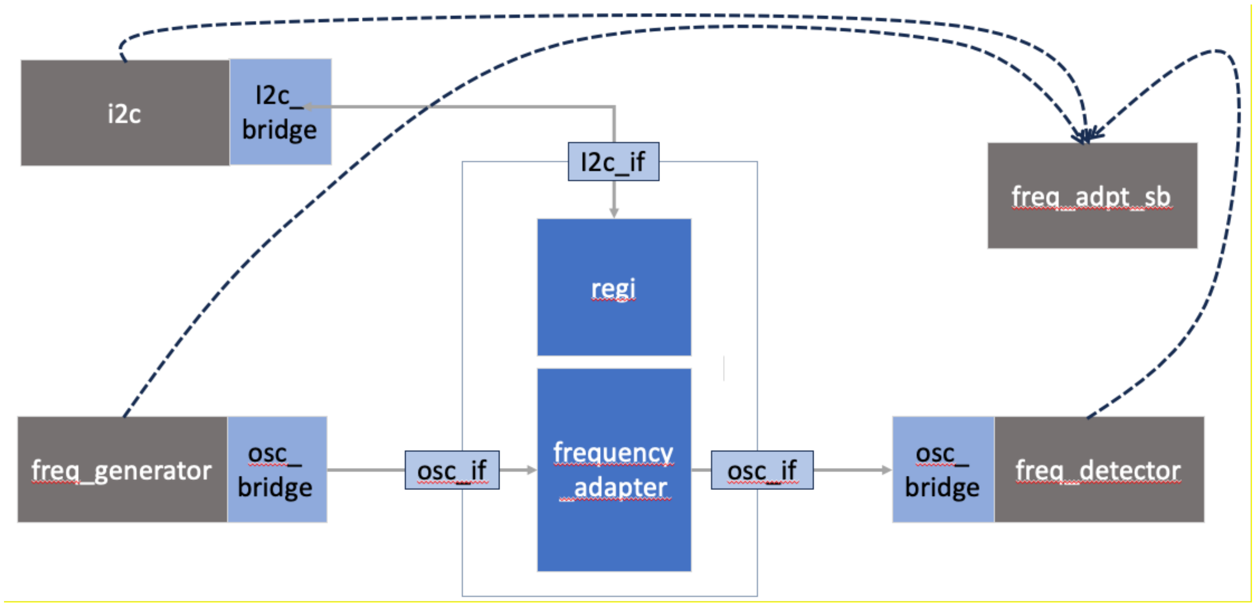


Figure 21 : UVM-MS testbench, showing oscillator bridges and interfaces

Annex C (informative) Bibliography

Brennan, J., T. Ziller, K. Fotouhi, and A. Osman, "The How To's of Advanced Mixed-Signal Verification," *DVCon Europe 2015*, accessed June 30, 2024, https://dvcon-proceedings.org/wp-content/uploads/the-how-tosof-advanced-mixed-signal-verification.pdf.

Freitas, A., and R. Santonja, "UVM Ready: Transitioning Mixed-Signal Verification Environments to Universal Verification Methodology," *DVCon Europe 2014*, accessed June 30, 2024, https://dvcon-proceedings.org/wp-content/uploads/uvm-ready-transitioning-mixed-signal-verification-environments-to-universal-verification-methodology.pdf.

Maurice, M., "Modeling Analog Devices using SV-RNM," *DVCon 2022*, accessed June 30, 2024, https://dvcon-proceedings.org/wp-content/uploads/Modeling-Analog-Devices-using-SV-RNM.pdf, https://dvcon-proceedings.org/wp-content/uploads/Modeling-Analog-Devices-using-SV-RNM-1.pdf.

Sanyal, S., A. Hazra, P. Dasgupta, S. Morrison, S. Surendran, and L. Balasubramanian, "CoveRT: A Coverage Reporting Tool for Analog Mixed-Signal Designs," *2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)*, Bangalore, India, 2020, pp. 119-124, accessed June 30, 2024, https://ieeexplore.ieee.org/document/9105529.

Vlach, M., and S. Little, "Tutorial: SystemVerilog-AMS: The Future of Analog/Mixed-Signal Modeling," *DVCon USA 2016*, accessed June 30, 2024, https://www.accellera.org/resources/videos/systemverilog-ams-tutorial-2016.