UVM-ML Integrator User Guide

For version UVM-ML 1.3

31 October, 2013

Notices

Questions or suggestions relating to this document or product can be sent to:
uvm_contributions@cadence.com

Table of Contents

Table of Diagrams

Overview

A Universal Verification Methodology-Multi-Language (UVM-ML) verification environment is constructed from multiple verification intellectual property components (IPs), which are implemented in different frameworks and communicate through the UVM-ML backplane.

For this user guide, it is sufficient for you to understand the basic concepts of UVM-ML, which are as follows:

- A standard backplane connects between the various frameworks.
- Each framework has an adapter interfacing between the backplane and the user code.

The UVM-ML architecture is described in detail in a separate document, titled: *UVM-ML Whitepaper*, which explains the role of the backplane and adapters when connecting multiple frameworks.

This user guide is primarily intended for integrators of such multi-language verification environments, but it also contains useful information for IP developers interested in ML environments.

For the integrator, it explains how to construct a multi-language verification environment using adapters and the UVM-ML backplane. The integrator views the project on the level of Frameworks and IP. The tools used for this are the adapter's interfaces connecting the various IPs. The backplane interface is hidden inside the adapters and is only mentioned as background information where relevant.

For end users of multi-language verification environments, this guide explains the main concerns and provides some information for the efficient use of the ML verification environment.

This guide is a part of the UVM-ML release package that contains three frameworks with the corresponding adapters, as well as several examples demonstrating the capabilities described here. While these are open source examples of adapters, they are intended to actually be used where they are appropriate, and also to serve as examples for the development of additional adapters in the future. The structure and content of the adapters are explained in a separate document titled: *Adapter Developer's Guide*.

Adapters

Each framework adapter identifies itself to the backplane with a unique identifier. Three adapters are included in the UVM-ML release:

- One for UVM-SV (for SystemVerilog and identified as *SV*)
- One for UVM-SC (for SystemC and identified as *SC*)
- One for UVM-e (for ***e*** and identified as *e*).

The adapters are registered at the backplane automatically, for example when they are instantiated in the framework.

The adapters have the following characteristics:

- The UVM-SV adapter is instantiated by adding the following lines in the SystemVerilog code, typically in the top-level component:

```
import uvm_pkg::*;
`include "uvm_macros.svh"
import uvm_ml::*;
```

Note: The UVM-SV adapter is built on top of the official release of the UVM-SV library. Some ML enablers were added in the original library, so for UVM-ML simulation you must use the modified version included in the UVM-ML release.

- The UVM-SC adapter is built on top of UVM-SC which is included in the UVM-ML release. This adapter is referred to by adding the following lines in the SystemC code of the top-level module (see examples how the adapter can be included in Linking and Simulating UVM-ML Environments below):

```
#include "uvm_ml.h"
using namespace uvm_ml;
```

- The UVM-***e*** adapter is included in this UVM-ML release. It is working on top of the Specman release in INCISIV12.20-s013 or later. The adapter is available for all users by default; no action is needed to make it available.

A unique identifier (ID) is assigned to each framework upon registration and is kept in the adapter to identify itself in future access to the backplane.

The adapters are registered at the backplane automatically upon instantiation, using the following API:

```
typedef int (*bp_register_framework_type)
(
   char *                  framework_name,          // name of the framework
   char *                  framework_indicators [], // identifiers for the framework
   bp_frmw_c_api_struct * framework_api             // address of the required API
);
```

Constructing an ML Testbench

An ML testbench is a testbench that is constructed from multiple frameworks. It can have one or more logical tops, each top representing a logical hierarchy (tree) in some framework. Each tree may contain sub-trees in multiple frameworks. The following diagram illustrates this.

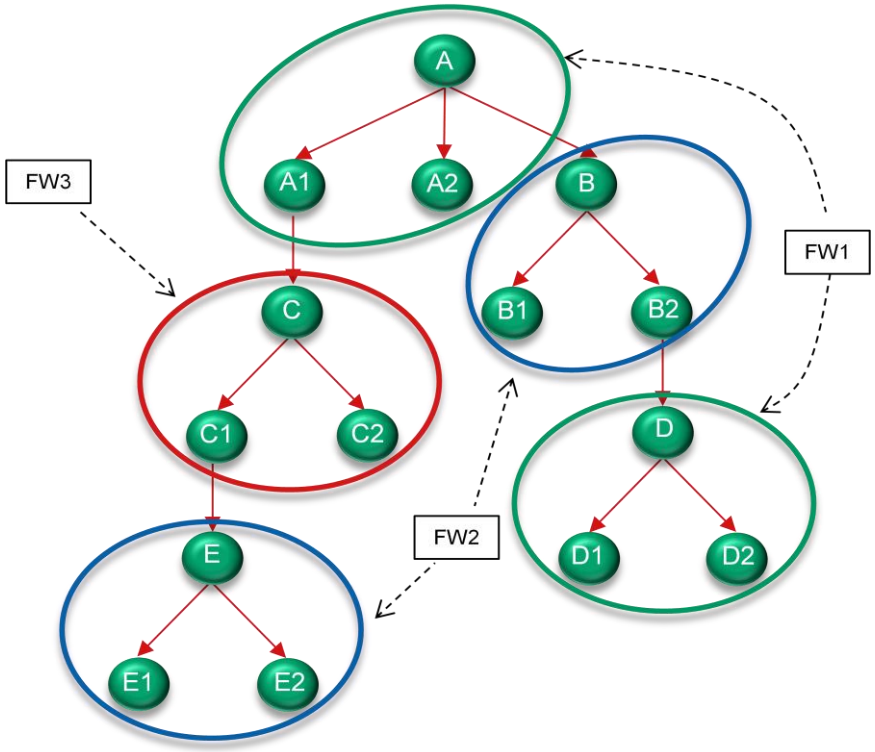

Diagram 1. Unified hierarchy in different frameworks

To create the ML testbench, you must declare the top components and the optional test component (which contains the variable parts that change from one simulation run to another). UVM-ML instantiates each top-component, and constructs the tree defined by it.

If a component has a child in another framework, the backplane is used to create the sub-tree in the appropriate framework and then propagate the phases between parent and child.

This section describes:

- How to instantiate a child component in a different framework
- How to construct a unified hierarchy out of multiple IP in different frameworks

Instantiating a Child Component in a Different Framework

The integrator is responsible for correctly specifying the type and instance name of the foreign component to be created in another framework. The type must be defined in the specified foreign framework.

The backplane will record and maintain the connection between the parent and foreign child frameworks, ensuring all phasing and other parent/child requests are correctly propagated. The following diagram illustrates this concept.

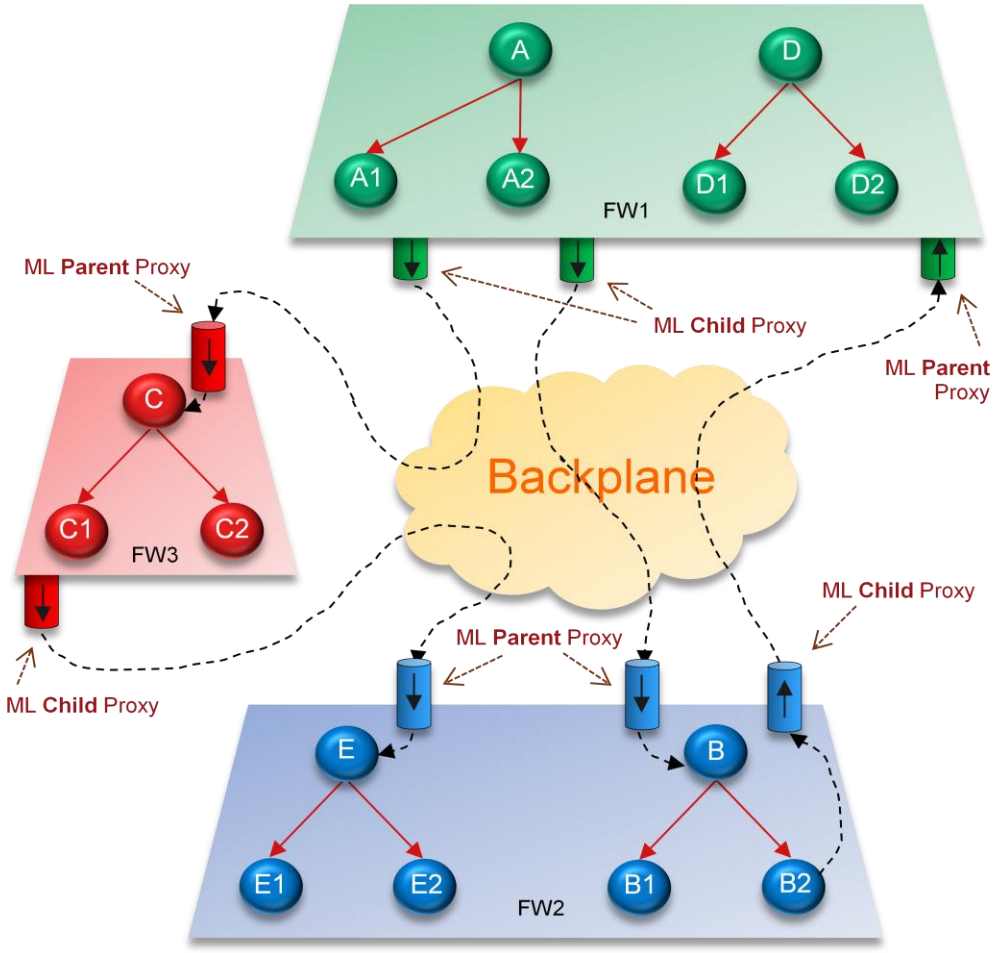

Diagram 2. Parent child relationship maintained through the backplane

The creation of a sub-tree in a different framework involves the creation of a child proxy, which in turn calls the backplane to request the construction of the foreign sub-tree. The backplane instructs the target framework to create a parent proxy with the requested sub-tree under it.

Once the connection is made, the backplane maintains the relationship between the child proxy and the parent proxy, for example for propagating the phases between child proxy and parent proxy, as shown in the diagram below.

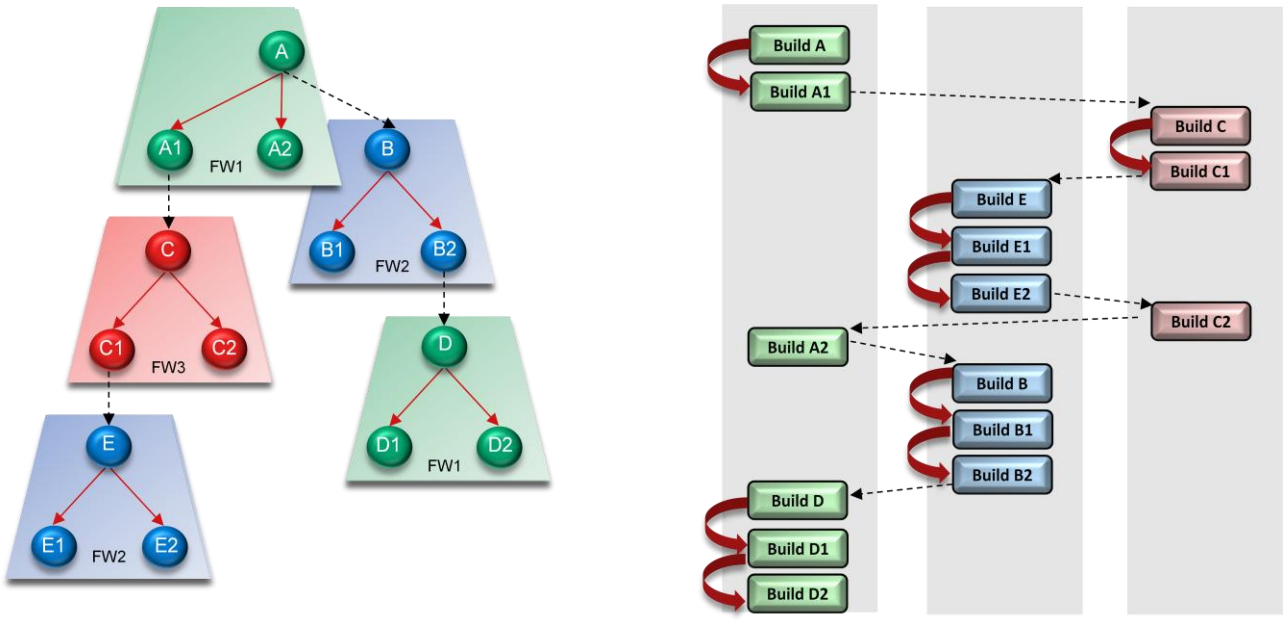

Diagram 3. Building and phasing scheme in unified hierarchy

The adapter's native interface for creating a sub-tree in a different framework is adjusted to the native language of the framework. The following list shows the adapter interface in the various languages:

- SV adapter API for creating a remote child:

```
function uvm_component uvm_ml_create_component(string        target_frmw_indicator,
                                                string        component_type_name,
                                                string        instance_name,
                                                uvm_component parent = null);
```

For example, creating a SC sub-tree of type `sctop` and instance name `sctop1`:

```
sc_child = uvm_ml_create_component("SC", "sctop", "sctop1", this);
```

- SC adapter API for creating a remote child:

```
child_component_proxy * uvm_ml_create_component(
                            const std::string   & target_frmw_indicator,
                            const std::string   & component_type_name,
                            const std::string   & instance_name,
                            const uvm_component * parent);
```

For example, creating a SV sub-tree of type `svtop` with instance name `svtop1`:

```
sv_child = uvm_ml_create_component("SV", "svtop", "svtop1", this);
```

- ***e*** API for creating a remote child:

```
instance_name: child_component_proxy is instance;
```

```
keep instance_name.type_name == string;
```

For example, creating an SC sub-tree of type `sctop` and the same instance name:

```
sc_child: child_component_proxy is instance;
      keep sc_child.type_name == "SC:sctop";
```

Note: When an e sub-tree is created under a remote component in another framework, the message control mechanism is limited, and recursive message settings (including default ones) done on sys might not apply to units in such a sub-tree. As a result, messages from those units might not be shown as expected on all destinations. As a workaround, you need to configure messages in such sub-trees separately.

The adapters use the backplane API to create a sub-tree. It is defined as follows:

```
typedef int (*bp_create_child_junction_node_type)
(
    int            framework_id,               // ID of the initiating framework
    const char   * target_framework_indicator, // name of the target framework
    const char   * component_type_name,        // type name of the foreign child
    const char   * instance_name,              // instance name of the foreign child
    const char   * parent_full_name,           // parent name
    int            parent_junction_node_id     // unique junction node ID
);
```

The backplane API to propagate phases to the sub-tree is as follows:

```
typedef int (*bp_transmit_phase_type)
(
    unsigned int      framework_id,    // ID of the initiating framework
    const char *      target_frmw_ind, // name of the target framework
    int               target_id,       // ID of the target junction node
    const char *      phase_group,     // name of the phase group
    const char *      phase_name,      // name of the phase
    uvm_ml_phase_action phase_action   // start, execute, ready to end, ended
);
```

This API is used internally in the adapters and is not visible in user code.

Configuring a Multi-Framework Testbench

UVM-ML broadcasts configuration values to other frameworks when the configuration value is set using the native configuration API. Within the various frameworks the values are maintained according to the framework's native behavior.

UVM-ML only propagates integral values, strings and serializable objects. For UVM-***e*** any struct is serializable. For UVM-SV and UVM-SC serializable objects are all objects derived from the uvm_object

base class. In UVM-SV, serializable object must have automation macros `uvm_field_* or be associated with an out-of-class serializer.

Configuration values that control the construction of the testbench are typically set and retrieved in the build phase so they can affect the testbench construction. The configuration during the build phase guarantees hierarchical precedence so a component higher in the unified hierarchy can override the setting of configuration values below it.

Wildcards can be used in the path and field name when setting configuration values. The interpretation of the wildcard is the responsibility of the target framework. UVM-ML recommends using '*' as a wildcard for arbitrary string. Other wildcards should be used carefully since some frameworks may not interpret them as expected. Currently Specman supports only '*'.

Configuration is implemented in a distributed manner. Configuration items are only propagated on the 'set' across the ML backplane, and all 'get' activity is retrieved from the local framework configuration 'cache' only.

Note: Currently the UVM-SV framework propagates in the UVM-ML environment only the configuration settings made in set_config_* and uvm_config_* methods. The methods of the parameterized class uvm_config_db#(T) currently do not broadcast the configuration settings to other frameworks.

Setting a Configuration Value

Each framework handles configuration according to its native behavior. When in a UVM-ML environment, the configuration data is propagated also to other frameworks according to the unified hierarchy of the testbench.

Setting configuration in UVM-SV

In UVM-SV use the native mechanism for setting configuration values. Setting a configuration value is done using the following function call, typically in the build phase:

```
set_config_*(inst_name, field_name, value);
```

or the alternative syntax

```
uvm_config_*(this, inst_name, field_name, value);
```

where '*' stands for 'int' for integral values, 'string' for string values and 'object' for objects.

For example setting the "address" field in all components named "producer" under this component, can be done as follows:

```
function void build_phase(uvm_phase phase);
   set_config_int("*producer","address",'h1000);
   super.build_phase(phase);
endfunction
```

Note: The ML-UVM configuration mechanism passes all integral values as uvm_bitstream_t. The getter of the configuration value must use proper casting to retrieve the value.

Setting configuration in UVM-SC

In UVM-SC setting a configuration value is done using the following function call, typically in the build phase:

```
set_config_*(inst_name, field_name, value);
```

where '*' stands for 'int' for integral values, 'string' for string values and 'object' for objects.

For example setting the "address" field in all components named "producer" can be done as follows:

```
void build() {
   set_config_int("*producer","address",'h1000);
}
```

UVM-SC currently supports string and object values as well as the following integral types: bool, char, short, ushort, int, uint, long, ulong, sc_dt::uchar, sc_dt::int64, sc_dt::uint64, sc_bv_base and sc_bv<>.

UVM-SC also supports the alternative methods

```
uvm_set_config_int("*producer", "address", 'h1000);
uvm_config_db<sc_bv_base>::set(this, "*producer", "address", 'h1000);
```

Setting configuration in UVM-e

In UVM-***e*** the build-time configuration value must be set in a declarative manner in randomization constraints to be able to affect the construction of the testbench. To set the value of a field in a sub-tree use the following:

```
keep uvm_config_set(inst_name, field_name, value);
```

For example to set a configuration value in all the components named "producer" use the following constraint:

```
unit testbench {
   keep uvm_config_set("*producer", "address", 0x1000);
};
```

Retrieving the Configuration Value

Configuration values are maintained locally in each framework such that the native methods can be used without intervention of the UVM-ML mechanisms.

Retrieving configuration in UVM-SV

In UVM-SV, configuration properties can be automated using the `uvm_field_* macros, and then they are retrieved automatically during the build phase. To get the configuration value manually use the following:

```
get_config_*(field_name, value);
```

where '*' stands for 'int' for integral values, 'string' for string values and 'object' for objects.

This will retrieve the configuration value for "field_name" and store it in "value". For example to get the value of "address" in the connect phase one can use the following:

```
function void connect_phase(uvm_phase phase);
      super.connect_phase(phase);
      get_config_int("address", address);
endfunction
```

When using get_config_object one must store the value in uvm_object and then cast to the appropriate object type

```
      uvm_object config_object;
      packet     config_packet;
      if ($cast(config_packet, config_object)) $display("Error");
```

UVM-SV supports also the alternative syntax e.g.:

```
      uvm_config_db#(uvm_bistream_t)::get()
      uvm_config_db#(uvm_object)::get()
```

Retrieving configuration in UVM-SC

In UVM-SC, to get the configuration value use the following:

```
   get_config_*(field_name, value);
```

with "int" for integral values, "string" for string values and "object" for objects.

This will retrieve the configuration value for "field_name" and store it in the "value" field. For example to get the value of "address" use the following:

```
void build() {
   get_config_int("address", address);
}
```

When the configuration value is an object, one must get it into uvm_object and then cast it to the proper object type:

```
    uvm_object *obj;
    packet *config_packet = NULL;
    get_config_object("config_packet", obj);
    if (obj != NULL) config_packet = DCAST<packet*> (obj);
```

UVM-SC currently supports string and object values as well as the following integral types: bool, char, short, ushort, int, uint, long, ulong, sc_dt::uchar, sc_dt::int64, sc_dt::uint64, sc_bv_base and sc_bv<>.

UVM-SC supports also the alternative syntax

```
    uvm_config_db<sc_bv_base>::get(this, "", "int", i);
    intval = i.to_int();
```

Retrieving configuration in UVM-e

UVM-***e*** uses the configuration values during the generation of the testbench, which is a declarative process, therefore the syntax is declarative. In UVM-***e***, to get the configuration value use

```
   keep uvm_config_get(value);
```

For example to get the configuration value of "address" in the "prod" unit one can use the following hard/soft constraint or conditional constraint:

```
unit prod {
    keep uvm_config_get(address); // get value if available
};
unit prod {
    keep soft uvm_config_get(address); // soft constraint can be overridden
};
unit prod {
    keep foo() => uvm_config_get(address); // get value if foo() returns true
};
```

Proper Usage and Limitations

- UVM-ML configuration mechanism propagates only integral, string and object values currently
- Arrays and vectors of non-singular types (with the exception of strings i.e. unpacked structures, unpacked arrays, and chandles) are not supported in the current version
- UVM-ML propagates the configuration values only when set using the supported interface functions described above. Changing the value by direct assignment is not reflected to other frameworks
- Build-time configuration settings are expected to be set in a parent component and affect the children components. Configuration setting made between siblings is not recommended because the precedence of the build-time settings is not defined for the components located at the same hierarchical level
- The combination of scope (full path) and field_name must be unique for proper UVM-ML propagation
- Wildcards may be interpreted differently in different frameworks
- Setting a configuration element from a static component (empty context, for example in a Verilog module) overrides all other settings for the same element
- UVM-SV and UVM-SC keep the configuration data per type while ***e*** has a common repository. Using multiple configuration elements with the same name but different type may not work well in UVM-ML.

- The first parameter of set_config_* describes the relative path to the configuration element, possibly with wildcard "*" representing an arbitrary string. UVM-***e*** does not support other wildcards like "?" so users should avoid them in UVM-ML environments.
- If there are multiple set_config_* applying to the same configuration element during the build phase, the value is taken from the highest component in the unified UVM-ML hierarchy. At run-time the last setting wins.
- In UVM-***e***, the value set in uvm_config_set() cannot depend on generated values in sub-units because they are generated after uvm_config_set() is executed.

Starting a Simulation Using a Multi-Framework Testbench

Starting a simulation can be done through a procedural interface defining the components in the code, or through a declarative interface, which is supported only in the Cadence Incisive Enterprise Simulator (IES).

Starting a Simulation Using a Procedural Interface

The UVM-SV API `uvm_ml_run_test()` replaces the SystemVerilog `run_test()`, or the SystemC `sc_start()`.The UVM-SV API must be called to indicate that the simulation is under control of UVM-ML, rather than a single language simulation.

The topmost components are specified as arguments of the SV adapter task `uvm_ml_run_test()`.

```
task uvm_ml_run_test(string tops[], string test = "");
```

where `tops` is a dynamic array of top component identifiers and `test` is an optional argument used to denote a test component. Each top and test component is identified by the framework and the type of the top component separated by a colon (**:**). It is assumed that the framework has the type defined and can be instantiated as a top component in the specified framework.

A test component has a predefined instance name, `uvm_test_top` (in accordance with UVM-SV). The fixed instance name enables re-use of the invariant hierarchical path when the test name is changing.

For example, to create a unified hierarchy with `my_test` as the top SystemVerilog test, use the following code:

```
module topmodule;
initial begin
   string tops[1];
   tops[0] = "";
   uvm_ml_run_test(tops, "SV:my_test");
end
```

If there is need to add more top components that are not under the test, then add them to the list of `tops`, as follows:

```
module topmodule;
```

```
initial begin
   string tops[2];
   tops[0] = "e:top.e";
   tops[1] = "SC:sctop";
   uvm_ml_run_test(tops, "SV:my_test");
end
endmodule
```

The example above shows an environment with two top components (one in ***e*** and one in SC) and a test component (in SV). The instance names of the top component are identical to the type name, while the instance name of the test component is always `uvm_test_top`.

Currently, there is no similar capability implemented in the other adapters.

Note: If the instance name of the top component is different from the type name, then you must provide it, appended after the type name, such as: `SV:svtop:my_top`. The default in the example above results in a top component with the instance name identical to the type name.

The backplane API used by the adapters for declaring the top and test components is shown below. When this API is called, UVM-ML verifies that the frameworks are available and registered. Then it instantiates the corresponding test and top components and builds them.

```
typedef int (*bp_run_test_type)
(
    int     framework_id, // ID of the initiating framework
    int     tops_n,       // number of top components
    char ** tops,         // list of top component type names
    char *  test          // name of the test component
);
```

Starting a Simulation Using the Declarative Method

IES provides an alternative method to define the top and test components using command- line declarations. This method uses the following command-line options:

- `uvmtop` declares a top-level component to be instantiated and built at the start of the simulation. This is the command line equivalent of “tops” in the procedural example above. One can have multiple uvmtop declarations on the command line.
- `uvmtest` declares a topmost test component to be instantiated and built at the start of the simulation. In UVM-SV and UVM-SC the test component instance name is always “uvm_test_top”. This is the command line equivalent of “test” in the procedural example above. One should declare only one uvmtest component which serves as the logical top of the hierarchy.

Note: -sctop should not be used for testbench components; it creates a static module instance which does not participate in the quasi-static phases.

Using this declarative method makes the procedural code, and calling `uvm_ml_run_test()` described above redundant. For example, the code above can be replaced by the following:

```
irun –uvmtest SV:my_test -uvmtop e:top.e –uvmtop SC:sctop …
```

Creating Multi-Language TLM connections

ML connections are based on TLM interfaces. The UVM-ML backplane contains a repository of connections that were designated as ML connections. It maintains the necessary information to route the transactions at run time from source to target(s).

It is the integrator's responsibility to implement these ML interfaces as described in the following sections:

- Registering ML connections
- Binding ML connections
- Rules for using ML connections at run time

The ML connections must adhere to TLM rules, meaning that both the *connector type* and the *transaction type* must match. Since the transactions are copied from the initiator to the target, the transaction type must be serializable and the adapters must have compatible serializer and de-serializer methods defined for the transaction type.

Registering ML connections

TLM ports and sockets used in ML connections must be registered in the backplane. Registering ML ports results in the adapters creating a channel for redirecting the transactions to and from the backplane. The adapters are responsible for creating the necessary infrastructure for the interface to the backplane.

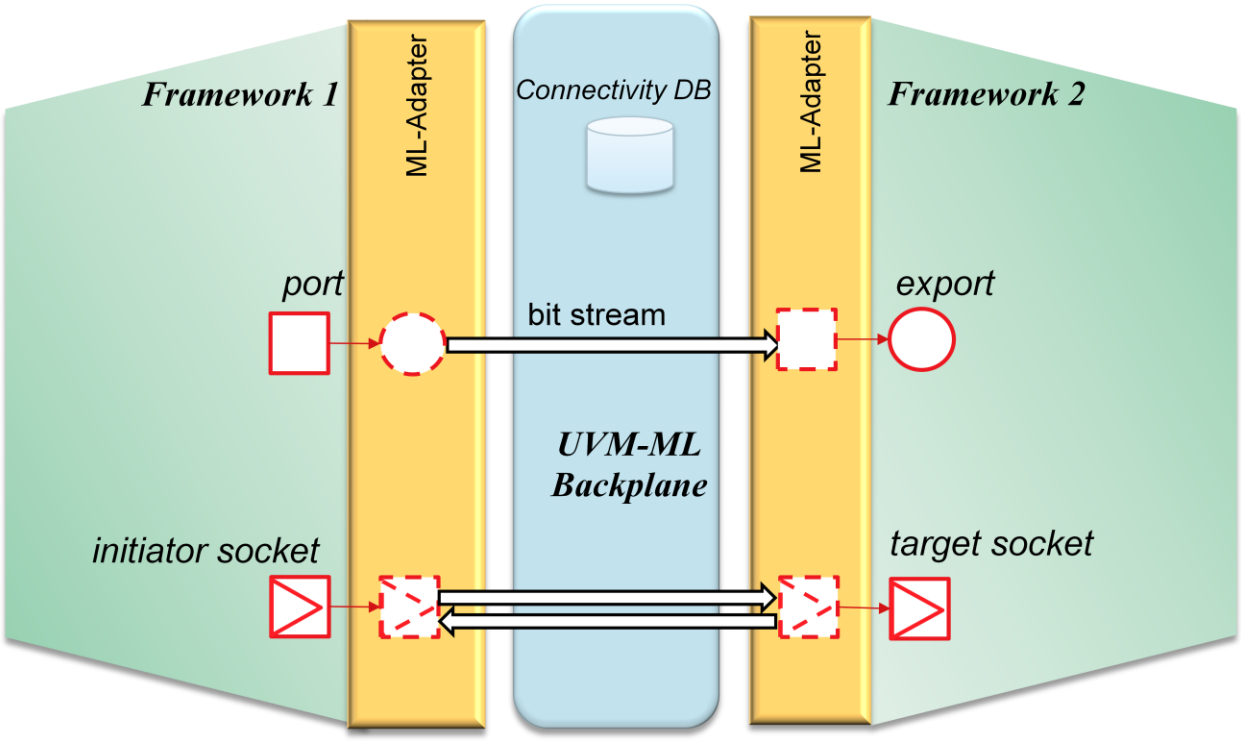

Diagram 4. TLM communication between frameworks

The adapter's native interface to register ML ports is adjusted to the specific language of the framework. The following examples show various combinations:

- The SystemVerilog API to register TLM1 ports and TLM2 sockets is as follows:

```
static function void register(uvm_port_base #(uvm_tlm_if_base #(T1,T2)) port,
                                                string T1_name="",
                                                string T2_name=""
);
```

The default for T2 is T1.

```
static function void register(uvm_port_base #(uvm_tlm_if #(TRAN_T,P)) sckt);
```

The default transaction type is uvm_tlm_generic_payload and the default type of phasing is uvm_tlm_phase_e.

For example, to register a TLM1 export and a TLM2 target socket:

```
function void phase_ended(uvm_phase phase);
   if (phase.get_name() == "build") begin
      uvm_ml::ml_tlm1#(packet)::register(sv_env.cons.put_export);
      uvm_ml::ml_tlm2::register(c.target_socket);
   end
endfunction
```

Note: The registration is executed in the `phase_ended()` callback of build because it is a suitable time point. It is called before the build phase is completed but after the child components with their sub-components and ports are built. The type for TLM2 transactions is not specified in the example above, so the default `uvm_tlm_generic_payload` is used.

- The SystemC API to register TLM1 port and TLM2 socket is as follows:

```
template <typename T, int N, sc_core::sc_port_policy POL>
   void uvm_ml_register(sc_core::sc_port<tlm::IF<T>,N,POL>* p);

template <class REQ, unsigned int BUSWIDTH, typename TYPES>
std::string ml_tlm2_register_initiator(const sc_module& containing_module,
                                       tlm_initiator_socket<BUSWIDTH,TYPES> &s,
                                       const std::string &initiator_socket_name,
                                       const std::string &cur_context_name);
template <class REQ, unsigned int BUSWIDTH, typename TYPES>
std::string ml_tlm2_register_target(const sc_module& containing_module,
                                    tlm_target_socket<BUSWIDTH,TYPES> &s,
                                    const std::string &target_socket_name,
                                    const std::string &cur_context_name)
```

For convenience, there are macros defined for TLM2 socket registration. The target socket above could be registered using the following macro:

```
ML_TLM2_REGISTER_TARGET(module_i,tran_t,sckt,buswidth)
```

For example, registering a TLM1 put port and a TLM2 initiator socket in the *before_end_of_elaboration* phase
(or in UVM-SC, you can use the *build* phase)

```
void before_end_of_elaboration() {
   uvm_ml::uvm_ml_register(&sc_env.prod.put_port);
   initiator_name = ML_TLM2_REGISTER_INITIATOR(i,tlm_generic_payload,isocket,32);
}
```

Note: The TLM2 registration macros are defined in `ml_tlm2.h`. The return value, `initiator_name` returns the full path to this socket to be used later, in a connect statement.

- ***e*** requires only that you declare the port or socket as `external`.

```
port: in interface_port of type of (transaction) is instance;
   keep bind(port, external);
```

For example, the registration of an analysis port of `packet` is done as follows:

```
inport: in interface_port of tlm_analysis of (packet) is instance;
   keep bind(inport, external);
```

Binding ML connections

To bind ports or sockets that were registered as ML connections, you must provide the full path of the port and export/import, or initiator socket and target socket, to the backplane. The backplane checks the compatibility of the two ends and registers the connection in its internal database.

The adapter's native interface for connecting ports and sockets is adjusted to the specific language of the framework. The following examples show various combinations:

- The SystemC API to connect a port to an export/import or an initiator to a target socket is as follows:

```
void uvm_ml_connect(
    const std::string & port_name,
    const std::string & export_name,
    bool map_transactions = true // option deprecated
);
```

For example, connecting an ***e*** port to an SC export:

```
uvm_ml::uvm_ml_connect("sys.u.outport", sc_env.cons.aexport.name());
```

- The SystemVerilog API to connect a port to an export/import or an initiator to a target socket is as follows:

```
function bit connect(string producer, string provider, bit map_transactions = 1);
```

For example, connecting an SV initiator socket to an SC target socket (`sc_consumer` is the path to the consumer in the SC framework):

```
res = uvm_ml::connect(p.initiator_socket.get_full_name(), {sc_consumer, "tsocket"})
```

- The ***e*** API to connect a port to an export/import or an initiator to a target socket is as follows:

```
uvm_ml.connect_names(string: initiator, string: target):bool;
```

For example, connecting an ***e*** port to an SC export (`sc_consumer` is the path to the SC *consumer* component):

```
res = uvm_ml.connect_names("sys.u.outport", append(sc_consumer,"aexport"));
```

Special Considerations for Non-Blocking TLM2 Transactions

The non-blocking transactions on the backward path are copied back to the same object that was sent on the forward path. This feature requires special attention:

- In SystemC, this mode requires the presence of a memory manager for the transaction (see section 14.5, Generic Payload Memory Management in the *IEEE Standard for Standard SystemC® Language Reference Manual 1666-2011.pdf*).
- If the end-user does not execute the full phasing protocol, and the transactions sent on the forward path never returns on the backward path, a memory leak will result.
- Currently, only `tlm_generic_payload` and types derived from it are supported by these ML sockets.

- The new field m_gp_option which was added to the tlm_generic_payload in the OSCI version 2.3 (see *1666-2011.pdf*) is not supported yet.

The adapters use the backplane API to connect ports or sockets. The API is as follows:

```
typedef unsigned (*bp_connect_type)
(
    unsigned       framework_id, // ID of the initiating framework
    const char *   path1,        // initiator path
    const char *   path2         // target path
);
```

Using the ML Connections

The calls to ML ports or sockets are the same as native calls, so the original IP code is not modified when the port or socket is bound to a different framework. UVM-ML passes transactions by a copy operation because the memory layout of the transaction may differ from one framework to another. The copying is implemented by means of serialization and de-serialization. Only the objects that can be serialized can be passed.

Since the data is copied rather than just passed by reference, at some point in time the testbench contains two copies of the data, which are synchronized only during handshake operations (*read*, *write*, or *acknowledge*). You cannot assume that any change in one copy is immediately reflected in the other copy. Also, garbage collection requires careful handling of the data.

Note: Since UVM-ML environments contain quasi-static components being created at time zero, you should not initiate any transactions at time zero. A race condition between simulators may result in inappropriate handling of blocking transactions that were initiated at time zero.

The adapters use the backplane API to send a transaction as follows (example TLM1 put):

```
typedef int (*bp_put_type)
(
    int                framework_id,      // ID of the initiating framework
    int                port_connector_id, // ID of the connector
    unsigned int       stream_size,       // size of the serialized data
    uvm_ml_stream_t    stream,            // the data (serialized)
    uvm_ml_time_unit * time_unit,         // simulation time (units)
    double           * time_value        // simulation time (value)
);
```

Note: the adapter adds the current time to each transaction. For more details see the Time Management section below.

Serialization

UVM-ML has a built in serializer for `tlm_generic_payload`. The serialization protocol is standardized and is described in the adapter developer's guide. The specific implementation of serialization is framework-specific. For example, the UVM-***e*** adapter implicitly serializes all transaction types. However, in UVM-SV and UVM-SC, you must explicitly describe serialization and de-serialization for each user-defined class. There are two kinds of serialization, which are explained below:

- In-class serialization
- Out-of-class serialization

If the type names in the initiator and target frameworks are not identical, you must declare the types as matching so the adapter can do the conversion between the types:

- The SystemVerilog API for mapping types is as follows:

```
function int set_type_match(string type1, string type2);
```

For example:

```
i = set_type_match "e:tlm::tlm_generic_payload" "sv:uvm_pkg::uvm_tlm_generic_payload";
```

- The ***e*** API for matching types is as follows:

```
uvm_ml_type_match type1 type2;
```

For example:

```
uvm_ml_type_match "e:tlm::tlm_generic_payload" "sv:uvm_pkg::uvm_tlm_generic_payload";
```

Note: The SystemC adapter currently does not have this feature implemented.

In-Class Serialization

In-class serialization defines the serialization method as part of the class definition:

- In-class serialization is supported in UVM-SV by defining the `uvm_object` macros. These macros generate the serialization code automatically. For example:

```
class packet extends uvm_object;
  int data;
  `uvm_object_utils_begin(packet)
    `uvm_field_int(data, UVM_ALL_ON)
  `uvm_object_utils_end
endclass
```

- In UVM-SC you must provide the `do_pack` and `do_unpack` methods explicitly, potentially taking advantage of the operator override >> for `unpack` and << for `pack`:

```
class packet : public uvm_object {
public:
  UVM_OBJECT_UTILS(packet)
```

```
  virtual void do_pack(uvm_packer& p) const {p << data;}
  virtual void do_unpack(uvm_packer& p) {p >> data;}
  int data;
};
```

- In ***e*** the serialization is handled automatically for all data types.

Out–Of-Class Serialization

Out-of-class serialization is used for types derived from `tlm_generic_payload`, or other places where in-class serialization is not defined.

The following example describes how to write a serializer class for a type `trans` derived from `tlm_generic_payload` with an extra integer field of `f0`.

In the UVM-SC adapter there are macros for generating the necessary serialization code. The code for these macros is available in `ml_tlm2.h`.

```
ML_TLM2_GP_BEGIN(trans)
   ML_TLM2_FIELD(f0)
ML_TLM2_GP_END(trans)
```

In the UVM-SV adapter the serializer class is defined as shown below. Later, its register function is called.

```
class ml_tlm2_trans_serializer extends uvm_ml::tlm_generic_payload_serializer;
  function string get_my_name();
    return "ml_tlm2_trans_serializer";
  endfunction
  function void serialize(uvm_object obj);
    trans inst;
    $cast(inst,obj);
    super.serialize(inst);
    pack_field_int(inst.f0, 32);
  endfunction
  function void deserialize(inout uvm_object obj);
    trans inst;
    super.deserialize(obj);
    $cast(inst,obj);
    inst.f0 = unpack_field_int(32);
  endfunction
  static function int register();
    static trans_serializer s;
    if (s == null) begin
      s = new;
```

```
      return uvm_ml::register_class_serializer(s, trans::get_type());
    end
    else
      return (-1);
  endfunction
endclass
```

The SystemVerilog API for registering a serializer class is as follows:

```
function bit register_class_serializer (uvm_ml_class_serializer serializer,
                                        uvm_object_wrapper      sv_type);
```

Known Limitations

The timing annotation argument in TLM2 blocking transport is supposed to be updated when the call is ended. Due to implementation issues the value of this argument is not being updated on return.

Time Management

The time value must be synchronized in all the frameworks at all times (except for un-timed frameworks, which may ignore the time). If a framework attempts to perform some action at a time that has already passed, an error is generated by the adapter.

Multiple Simulators

When multiple simulators are involved in the simulation of a verification environment, one simulator must be designated as the *time master*, while all other simulators work in *slave mode*. Slave simulators must adjust their time to the time of the master, and they should never advance the time beyond the master simulator's time.

The master simulator is the simulator activated by the command-line when the simulation run is started. Slave simulators can be linked into the code like the OSCI SystemC simulator is, as shown in the UVM-ML with Incisive and OSCI SystemC section below.

Synchronizing Simulators

Simulators are synchronized on every ML transaction. Transactions initiated by the master simulator propagate the time to the slave simulators. If however, a slave simulator initiates a transaction, it must wait for the master simulator to advance the time before it can initiate the transaction. This can be achieved by a synchronization loop in the master simulator which broadcasts the time to all the frameworks using the backplane API, as described below.

The SystemVerilog API for synchronizing slave simulators is as follows:

```
  task synchronize();
```

The example below shows the master simulator (IES or VCS) working with OSCI as the slave simulator. The `uvm_ml::synchronize()` task is called to propagate the timing information to the backplane, which in turn broadcasts it to the rest of the frameworks.

```
class svtop extends uvm_test;
    `uvm_component_utils(svtop)
    function new(string name, uvm_component parent=null);
        super.new(name, parent);
    endfunction
    task run_phase(uvm_phase phase);
        while(1) begin
            #1 uvm_ml::synchronize();
        end
    endtask
endclass
```

Note: the strobe rate should be set to the minimal needed frequency to minimize the runtime performance overhead.

The adapters use the backplane API to synchronize as follows:

```
typedef void (*bp_synchronize_type)
(
    int               framework_id, // ID of the initiating framework
    uvm_ml_time_unit  time_unit,    // simulation time (units)
    double            time_value    // simulation time (value)
);
```

Services

Services can be provided by various frameworks. This feature is not currently implemented, except for a partial implementation of phasing.

Phasing Master

Phasing is a service that can be provided by any framework. Currently there is no way to change the phasing service. When the UVM-SV adapter is used, it registers itself as the phasing master. If it is not instantiated, the default phase master in the backplane is used.

Linking and Simulating UVM-ML Environments

To use UVM-ML, you must link all the components (backplane, adapters, and so on) and activate the simulator with the correct flags.

The UVM-ML release package contains Makefiles for IES, VCS and Questa in ml/tests. To use them you must provide some parameters. Follow the examples in the ml/examples/demos directory. The provided Makefiles have multiple targets:

- make ies – to run with IES when the top level components are defined with uvm_ml_run_test()

- make ies_ncsc_cl – to run with IES when the top components are declared on the command line
- make ies_osci – to run with IES and OSCI SystemC simulator
- make vcs – to run with VCS
- make questa – to run with Questa
- make clean – to remove the results of the simulation except for the log files

Several controls are available to modify the behavior of the makefiles, for example you can run the simulation in GUI mode by adding GUI_OPT="-gui" EXIT_OPT="".

The following environment variables can be used also to enhance the function of the makefiles:

- EXTRA_IRUN_ARGS
  To add argument to irun e.g. setenv EXTRA_IRUN_ARGS "-enable_DAC"
- EXTRA_OSCI_ARGS
  To add arguments to OSCI simulator compilation
- EXTRA_SN_COMPILE_ARGS (Makefile.vcs / Makefile.questa)
- EXTRA_VCS_ARGS (Makefile.vcs)
- EXTRA_VSIM_ARGS (Makefile.questa)

Alternatively you can compile and link your code and run the various tools as shown below for some specific examples.

Running a UVM-ML Simulation with the Incisive irun Utility

To run a simulation with two frameworks (UVM-SC and UVM-SV) using *irun*, issue the following command:

```
irun -64bit -ml_uvm \
./svtop.sv ./sctop.cpp \
-top topmodule \
-f $UVM_ML_HOME/ml/tests/irun_uvm_ml.64.f\
-uvmhome $UVM_ML_HOME/ml/frameworks/uvm/sv/uvm-1.1c \
-sysc -DSC_INCLUDE_DYNAMIC_PROCESSES -scsynceverydelta on \-
I$UVM_ML_HOME/ml/adapters/uvm_sc \
-I$UVM_ML_HOME/ml/adapters/uvm_sc/common \
-I$UVM_ML_HOME/ml/adapters/uvm_sc/ncsc
```

The example above:

- The second line lists the source files and the top module, which declares the top components
- –top topmodule" declares the top level SV module which contains uvm_ml_run_test().
- For UVM-SC you must define SC_INCLUDE_DYNAMIC_PROCESSES
- -uvmhome must point to the ML-ready version included in the UVM-ML release

If ***e*** is involved, one must add:

```
./etop.e
-snshlib $UVM_ML_HOME/ml/libs/uvm_e/12.2/64bit/libsn_sn_uvm_ml.so
```

Using the Makefile

Alternatively when using the makefiles included in the release, create the following local Makefile for your environment as follows:

```
SRC_FILES = ./svtop.sv
SYSC_FILES = ./sctop.cpp
HAS_E = 1
E_FILES =./etop.e
SV_TOP = -top topmodule
include $(UVM_ML_HOME)/ml/tests/Makefile.ies
```

Here SV_TOP declares the top level SV module that contains uvm_ml_run_test().

Invoke the makefile as follows:

```
make ies
```

Running a UVM-ML Simulation on IES with OSCI SystemC

To use the OSCI SystemC simulator instead of IES, do the following:

1. First, compile the SystemC code and create a combined library with the OSCI simulator, the ML-UVM library, and the adapter and framework libraries:

   ```
   g++ -fPIC -o liball_osci.64.so ./sc.cpp \
   -I$OSCI_SRC \
   -I$TLM2_INSTALL \
   -I$UVM_ML_HOME/ml/adapters/uvm_sc/ \
   -I$UVM_ML_HOME/ml/adapters/uvm_sc/osci \
   -I$UVM_ML_HOME/ml/frameworks/uvm/sc \
   -Xlinker -rpath -Xlinker $UVM_ML_HOME/ml/libs/osci/2.2/64bit \
   -L$UVM_ML_HOME/ml/libs/osci/2.2/64bit \
   -luvm_sc_fw_osci -luvm_sc_ml_osci –shared
   ```

2. Next, you can use *irun* to run the simulation, by providing the combined SystemC library from the previous step:

   ```
   irun -64bit -ml_uvm \
   ./test.sv -top topmodule \
   -f $UVM_ML_HOME/ml/tests/irun_uvm_ml.64.f \
   -uvmhome $UVM_ML_HOME/ml/frameworks/uvm/sv/uvm-1.1c \
   -snshlib $UVM_ML_HOME/ml/libs/uvm_e/64bit/libsn_sn_uvm_ml.so
   -L`pwd` liball_osci.64.so
   ```

Using the Makefile

Alternatively when using the makefiles included in the release, use the same Makefile described in the previous section, and invoke it as follows:

```
make ies_osci_proc
```

Running a UVM-ML Simulation with VCS and Specman

To use the VCS simulator and Specman in interpreted mode, do the following:

1. Create Specman SV stub file for VCS (specman.sv):

   ```
   specman -c "load top.e; write stubs -vcssv"
   ```

2. Compile the SystemVerilog files, including Specman stub file. You have to make sure all your e files are given as arguments to uvm_ml_run_test():

   ```
   setenv VCS_UVM_HOME $UVM_ML_HOME/ml/frameworks/uvm/sv/uvm-1.1c/src
      vcs -o simv64 \
      -CFLAGS ' -O0 -g' -LDFLAGS ' -O0 -g -Wl,-E' \
      -sverilog +acc +vpi -timescale=1ns/1ns \
      -ntb_opts uvm-1.1 \
      -full64 \
      +define+CDNS_EXCLUDE_UVM_EXTENSIONS +define+CDNS_RECORDING_SVH \
      +incdir+$VCS_UVM_HOME/seq \
      +incdir+$VCS_UVM_HOME \
      +incdir+$VCS_UVM_HOME/tlm2 \
      +incdir+$VCS_UVM_HOME/base \
      +incdir+. \
      +incdir+$UVM_ML_HOME/ml/adapters/uvm_sv \
      $UVM_ML_HOME/ml/adapters/uvm_sv/uvm_ml_adapter.sv \
      test.sv specman.sv \
      $UVM_ML_HOME/ml/libs/backplane/64bit/libml_uvm.so \
      -P `sn_root -home`/src/pli.tab \
      `sn_root -dir -64`/libvcssv_sn_boot.so
   ```

3. Set SPECMAN_DLIB to load the precompiled UVM-***e*** adapter:

   ```
   setenv SPECMAN_DLIB $UVM_ML_HOME/ml/libs/uvm_e/64bit/libsn_sn_uvm_ml.so

   setenv SPECMAN_PATH $UVM_ML_HOME/ml/libs/uvm_e/64bit
   ```

4. Finally, to run the simulation, the image generated by VCS is launched:

   ```
   ./simv64
   ```

To use VCS simulator and Specman in compile mode, do the following:

1. Compile your ***e*** files on top of UVM-***e*** adapter:

```
sn_compile.sh -64 \
-s $UVM_ML_HOME/ml/libs/uvm_e/64bit/sn_uvm_ml \
./top.e  -o libsn_top.so -shlib
```

2. Create Specman SV stub file for VCS (specman.sv):

```
specman -c "load top.e; write stubs –vcssv"
```

3. Compile the SystemVerilog files, including Specman stubs file:

```
vcs -o simv64 \
    -CFLAGS ' -O0 -g' -LDFLAGS ' -O0 -g -Wl,-E' \
    -sverilog +acc +vpi -timescale=1ns/1ns \
    -ntb_opts uvm-1.1 \
    -full64 \
    +define+CDNS_EXCLUDE_UVM_EXTENSIONS +define+CDNS_RECORDING_SVH \
    +incdir+$VCS_UVM_HOME/seq \
    +incdir+$VCS_UVM_HOME \
    +incdir+$VCS_UVM_HOME/tlm2 \
    +incdir+$VCS_UVM_HOME/base \
    +incdir+. \
    +incdir+$UVM_ML_HOME/ml/adapters/uvm_sv \
    $UVM_ML_HOME/ml/adapters/uvm_sv/uvm_ml_adapter.sv \
    test.sv specman.sv \
    $UVM_ML_HOME/ml/libs/backplane/64bit/libml_uvm.so \
    -P `sn_root –home`/src/pli.tab \
    `sn_root –dir -64`/libvcssv_sn_boot.so
```

4. Set SPECMAN_DLIB to load the precompiled e code:

```
setenv SPECMAN_DLIB ./libsn_top.so
```

5. Finally, to run the simulation, the image generated by VCS is launched:

```
./simv64
```

Running a UVM-ML Simulation with VCS and OSCI

To use the OSCI SystemC simulator with VCS, do the following:

1. First, compile the SystemC code and create a combined library with the OSCI simulator, the ML-UVM library, and the adapter and framework libraries:

```
`ncroot`/tools/cdsgcc/gcc/bin/64bit/g++ -g -fPIC -o liball_osci.64.so sc.cpp \
-I$OSCI_SRC \
-I$TLM2_INSTALL \
-I$UVM_ML_HOME/ml/adapters/uvm_sc \
-I$UVM_ML_HOME/ml/adapters/uvm_sc/osci \
-I$UVM_ML_HOME/ml/frameworks/uvm/sc \
-Xlinker -rpath -Xlinker $UVM_ML_HOME/ml/libs/osci/2.2/64bit \
-Xlinker -rpath -Xlinker $UVM_ML_HOME/ml/libs/backplane/64bit \
-L$UVM_ML_HOME/ml/libs/osci/2.2/64bit \
-L$UVM_ML_HOME/ml/libs/backplane/64bit \
-lml_uvm -luvm_sc_fw_osci -luvm_sc_ml_osci -shared
```

2. Next, VCS is invoked to compile the SystemVerilog files and link them with the library created above:

```
setenv VCS_UVM_HOME $UVM_ML_HOME/ml/frameworks/uvm/sv/uvm-1.1c/src
vcs -o simv64 \
-CFLAGS ' -O0 -g' -LDFLAGS ' -O0 -g -Wl,-E' \
-sverilog +acc +vpi -timescale=1ns/1ns \
-ntb_opts uvm-1.1 \
-full64 \
+define+CDNS_EXCLUDE_UVM_EXTENSIONS +define+CDNS_RECORDING_SVH \
+incdir+$VCS_UVM_HOME/seq \
+incdir+$VCS_UVM_HOME \
+incdir+$VCS_UVM_HOME/tlm2 \
+incdir+$VCS_UVM_HOME/base \
+incdir+. \
+incdir+$UVM_ML_HOME/ml/adapters/uvm_sv \
$UVM_ML_HOME/ml/adapters/uvm_sv/uvm_ml_adapter.sv test.sv \
$UVM_ML_HOME/ml/libs/backplane/64bit/libml_uvm.so \
liball_osci.64.so
```

3. Finally, to run the simulation, the image generated by VCS is launched:

```
./simv64
```

Working with UVM-*e* Adapter

The UVM-***e*** adapter is part of the UVM-ML package and therefore requires being loaded in each test where UVM-***e*** is present. In the above examples there are several ways of doing that, depending on the simulator and the desired flow.

The following are the suggested ways:

- **irun** – the UVM-***e*** adapter should be loaded with –snshlib <UVM-***e*** adapter library>.

Additional ***e*** test files should appear on the command line with –uvmtop/-uvmtest irun options or provided using uvm_ml_run_test().

- **VCS** –the UVM-***e*** adapter can be loaded by setting SPECMAN_DLIB to point to the UVM-***e*** adapter library and SPECMAN_PATH to the UVM-***e*** adapter directory.
  Additional ***e*** test files can be loaded with uvm_ml_run_test() or by compiling the test files on top of the UVM-***e*** adapter executable to a shared library and pointing SPECMAN_DLIB to it.

Note: To save the need to load the UVM-***e*** adapter in each test, one may use **sn_enanace.sh** to update your current Specman version to include the UVM-***e*** adapter. This step should be taken only after you succeed to run several UVM-***e*** tests.

Appendix I: User API functions

This Appendix lists the user API functions.

UVM-SV Adapter

TLM Interface

- Register ML socket.

```
static function void register(uvm_port_base #(uvm_tlm_if #(TRAN_T,P)) sckt);
```

- Register ML port/export.

```
static function void register(uvm_port_base #(uvm_tlm_if_base #(T1,T2)) port,
                                               string T1_name="",
                                               string T2_name="");
```

- Connect port/initiator socket to export/target socket.

```
function bit connect(string producer_name,
                     string provider_name,
                     bit map_transactions = 1);
```

- Register serializer.

```
function int register_tlm_generic_payload_serializer();
function bit register_class_serializer (uvm_ml_class_serializer serializer,
                                        uvm_object_wrapper      sv_type);
```

- Indicate matching types (when the type has different names in different frameworks) .

```
function int set_type_match(string type_name1, string type_name2);
```

- Map return data back to the original data structure.

```
function void set_transaction_mapping(string socket_name);
```

Unified Hierarchy

- Create a sub-tree in a remote framework.

```
function uvm_component uvm_ml_create_component(string    target_framework_indicator,
                                               string      component_type_name,
                                               string      instance_name,
                                               uvm_component parent = null);
```

Configuration

- Set configuration value

```
set_config_*(inst_name, field_name, value);
```

- Get configuration value

```
get_config_*(field_name, value);
```

Simulation Control

- Synchronize all simulators (for example, the Incisive Enterprise Simulator and OSCI).

```
task synchronize();
```

- Start an ML simulation.

```
task uvm_ml_run_test(string tops[], string test = "");
```

UVM-SC Adapter

The API is implemented in the namespace `uvm_ml`. It has the following public methods.

TLM Interface

- Register ML port/export.

```
template <typename T, int N, sc_core::sc_port_policy POL>
   void uvm_ml_register(sc_core::sc_port<tlm::IF<T>,N,POL>* p);
template <typename T>
    void uvm_ml_register(sc_core::sc_export<tlm::IF<T> >* p);
```

- Register ML socket.

```
template <class REQ, unsigned int BUSWIDTH, typename TYPES>
std::string ml_tlm2_register_initiator(const sc_module& containing_module,
                                       tlm_initiator_socket<BUSWIDTH,TYPES> *s,
                                       const std::string &initiator_socket_name,
                                       const std::string &cur_context_name);
template <class REQ, unsigned int BUSWIDTH, typename TYPES>
std::string ml_tlm2_register_initiator(const sc_module& containing_module,
                                       tlm_initiator_socket<BUSWIDTH,TYPES> &s,
                                       const std::string &initiator_socket_name,
                                       const std::string &cur_context_name);
   template <class REQ, unsigned int BUSWIDTH, typename TYPES>
   std::string ml_tlm2_register_target(const sc_module& containing_module,
```

```
                                        tlm_target_socket<BUSWIDTH,TYPES> *s,
                                        const std::string &target_socket_name,
                                        const std::string &cur_context_name);
    template <class REQ, unsigned int BUSWIDTH, typename TYPES>
    std::string ml_tlm2_register_target(const sc_module& containing_module,
                                        tlm_target_socket<BUSWIDTH,TYPES> &s,
                                        const std::string &target_socket_name,
                                        const std::string &cur_context_name);
```

- Macros for registering sockets.

```
ML_TLM2_REGISTER_TARGET(module,tran_t,sckt,buswidth)
ML_TLM2_REGISTER_TARGET_WITH_PROTOCOL(module,tran_t,sckt,buswidth,proto_types)
ML_TLM2_REGISTER_INITIATOR(module,tran_t,sckt,buswidth)
ML_TLM2_REGISTER_INITIATOR_WITH_PROTOCOL(module,tran_t,sckt,buswidth,proto_types)
```

- Connect port/initiator socket to export/target socket.

```
void uvm_ml_connect(
  const std::string & port_name,
  const std::string & export_name,
  bool map_transactions = true // deprecated
);
```

Unified Hierarchy

- Create a sub-tree in a remote framework.

```
child_component_proxy * uvm_ml_create_component(
    const std::string   & target_framework_indicator,
    const std::string   & component_type_name,
    const std::string   & instance_name,
    const uvm_component * parent
);
```

Configuration

- Set configuration value

```
set_config_*(inst_name, field_name, value);
```

- Get configuration value

```
get_config_*(field_name, value);
```

UVM-e Adapter

TLM Interface

- Connect port/initiator socket to export/target socket.

```
uvm_ml.connect_names(string: initiator, string: target):bool;
```

Unified Hierarchy

- Create a sub-tree in a remote framework.

```
instance_name: child_component_proxy is instance;
   keep instance_name.type_name == string;
```

Configuration

- Set configuration value

```
keep uvm_config_set(inst_name, field_name, value);
```

- Get configuration value

```
keep uvm_config_get(value);
```

Appendix II: Incisive Enterprise Simulator Command-Line Options

IES simulation is invoked using *irun*. The `irun` command has many options, which can be listed using the `irun -helpall` command. This appendix explains the relevant flags used for UVM-ML.

- `-64bit` : Invoke the 64-bit version of *ncelab*.
- `-ml_uvm` : Enable multi-language UVM-ML.
- `-top <topmodule>` : Specify the top-level unit or module for the simulator.
- `-uvmtop <testName>` : Specify the top test class name.

  Use this flag only with IES when using command-line declarations of the top components, instead of `uvm_ml_run_test`. A top-level component will be instantiated with the instance name identical to the type name.

- `-uvmtest <testName>` : Specify the test class name .

  Use only with IES when providing the test name on the command line instead of uvm_ml_run_test. It will define the class to be used as the test and the instance name will always be `uvm_test_top`.

- `-uvmhome <path>` : Location to look for the UVM-SV install.
- `-sysc` : SystemC is present.

  Use only with native IES SystemC, and not with OSCI. When OSCI SystemC is used, it is linked in as a library rather than *irun* invoking it.

- `-sv_lib <library>` : Dynamically load a DPI library.

  Use for linking in a shared DPI library, such as in the case of the backplane.

- `-snsv` : Specify SystemVerilog agent for Specman.

  Indicate to Specman that the simulator is SystemVerilog, in order to generate the appropriate stub. Use only if ***e*** code is involved.

- `-snsc` : Specify SystemC agent for Specman.

  Indicate to Specman that the simulator is SystemC, in order to generate the appropriate stub. Use only if ***e*** code is involved.

- `-snshlib <library>`: Load the precompiled ***e*** code
  Use for loading the UVM-***e*** adapter library.

- `-D<>` : Set define value for SystemC, for example: `-DSC_INCLUDE_DYNAMIC_PROCESSES`.
- `-I<path>` : Set search path for SystemC header files.
- `-L<path>` : Set search path for libraries.

- `-Wld,-Xlinker -Wld,-rpath -Wld,-Xlinker -Wld,<path>` : Specify the runtime library search path for the linker.
- `-Wcxx,-I<path>` : Set header search path to C++ compiler.

Note: Before running any simulations, you must source the `setup.sh` script which prepares the necessary libraries and sets some additional environment variables used in the various *makefiles* and demo scripts.