cadence®

UVM-ML OA User Guide

Product Version 1.5.1

Aug 2015

Copyright © 2013-2015 Cadence Design Systems, Inc. All rights reserved.
Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Copyright © 2013-2015 Advanced Micro Devices, Inc. (AMD). All rights reserved.
Advanced Micro Devices, Inc., One AMD Place, P.O. Box 3453, Sunnyvale, CA 94088, USA.

This product is licensed under the Apache Software Foundation's Apache License, Version 2.0 (the "License") January 2004. The full license is available at http://www.apache.org/licenses/LICENSE-2.0 .

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Notice

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

Contents

1

About This Book

This guide describes creating and using multi-language (ML) verification environments using the UVM Multi-Language Open Architecture package (UVM-ML). Multi-language verification environments are environments in which components are implemented in more than one verification language.

If this is your first ML project, we recommend starting with the Common Use Models of Multi Language Environment chapter, which contains descriptions of the most common ML use models. Each section explains and demonstrates the steps required for creating the required environment.

If you know the task you need to perform, you can go directly to the task of interest.

The last chapter of this guide, Troubleshooting , discusses common issues and suggested solutions.

The following use models are described in this document:

1. Integrating a SystemVerilog Component into an e Verification Environment
2. Integrating an e Unit into a SystemVerilog Verification Environment
3. Integrating a Reference Model Implemented in SystemC into an e Verification Environment
4. Integrating a Reference Model Implemented in SystemC into a SystemVerilog Verification Environment
5. Connecting Monitors of Various Languages, to One Checker

The following verification tasks are described in this document:

- Creating a Multi-Language Environment
- Configuration in a Multi-Language Environment
- Compiling and Running Multi-Language Environments
- Multi-Language Data Communication
- Do'ing Sequences Implemented in Another Language
- Ending (Stopping) The Test
- Debugging in a Multi-Language Environment

See Also

- The "UVM-ML Key Concepts" chapter in the *UVM-ML OA Reference* provides an overall description of the UVM-ML underlying methodology and architecture.

2

Common Use Models of Multi Language Environment

This chapter describes how to create a multi-language verification environment.

Each sub-chapter describes the creation of one use model.

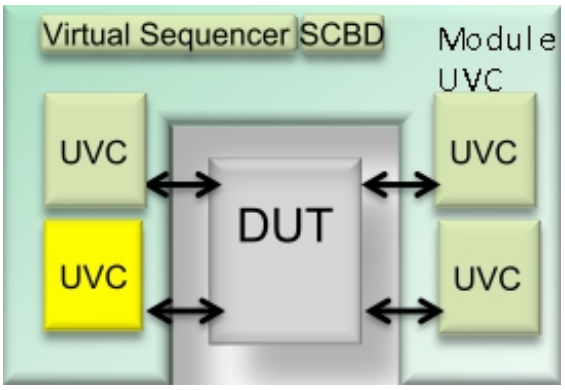
In this chapter:

Integrating a SystemVerilog Component into an e Verification Environment

This section describes how to integrate a SystemVerilog component into an ***e*** verification environment.

This section assumes that you are already familiar with the main principles of creating a system-level environment, all of which also apply to creating multi-language environments. This section focuses instead on the integration of UVCs implemented in different languages.

Following is an illustration of the basic model described in this section.



Following are the basic steps for integrating a SystemVerilog component into an ***e*** verification environment. Each step is described in detail in the sections that follow in this guide:

1) Build the environment hierarchy

You can mix verification environments developed in different languages with UVM-ML. The components can be instantiated in a unified hierarchy or in a side-by-side hierarchy (parallel trees).

- Unified hierarchy--each component is instantiated in its logical location in the hierarchy
  - For example, in an environment similar to the one shown in the picture above, four sub UVCs--three implemented in ***e*** and one implemented in SystemVerilog--would be instantiated in one module env.
  - For more details, see Instantiating a SystemVerilog Component Within an ***e*** Unit .
- Side-by-side hierarchy (parallel trees)--the environment contains multiple tops, each top containing the components of one language.
  - For example, in an environment similar to the one shown in the picture above, there would be two tops. The ***e*** top, **sys** , would contain the module env (which would contain three sub UVCs), and another top, a SystemVerilog top component, would contain the SystemVerilog sub UVC.
  - For details, see Creating a Side-by-Side (Multiple Tops) Environment .

> For more information about unified hierarchies and side-by-side hierarchies, including illustrative figures, see "UVM-ML Hierarchy" in the "UVM-ML Key Concepts" chapter of the *UVM-ML OA Reference*

2) Configure the sub UVC

In a unified hierarchy, you can configure the SystemVerilog sub UVC from the ***e*** UVC. For example, you could configure the number of agents to construct or the UVC work mode. (Typical configuration tasks are described in e Units Configuring SystemVerilog Components .)

When working in a side-by-side hierarchy, each components is configured separately, as in single language environments, and you have to manually take care that the configuration align.

3) Connect monitors

If any of the data collected by the SystemVerilog components is to be used in the ***e*** components, for example by a system level checker implemented in the ***e*** UVC, you need to instantiate input ports in the ***e*** component and connect them to the SystemVerilog monitor ports. See SystemVerilog Components Passing Data to e Units .

4) Add multi-language sequences

Add multi-language sequences as per your test requirements. For example, if you want a virtual sequence implemented in ***e*** to **do** sequences of the SystemVerilog UVC, you can use multi-language sequences. See ML Sequences - e Over SystemVerilog .

5) Write tests

In ***e*** , a test is implemented in an ***e*** file that configures the verification environment behavior per test definition. A typical ***e*** test file contains constraints directing the generation and perhaps definitions of sequences (by extending the MAIN sequence **body** ()).

When the environment contains components in SystemVerilog, there are two main approaches to creating the ***e*** tests:

- Independent behavior
- Central control

The "independent behavior" approach

This approach is applicable when the SystemVerilog and ***e*** parts need not be coordinated during the test. For example, this approach is appropriate if the SystemVerilog UVC contains one reactive agent that responds to requests coming from the DUT. In such a case, you define the behavior of the SystemVerilog component as you would in a SystemVerilog-only environment.

Note that each run (that is, simulation) should have only one **uvm test.** If the **uvm test** is the ***e*** file, you should not instantiate a SystemVerilog **uvm_test** component. Rather, the behavior of the SystemVerilog components should be controlled from the top component, the **uvm_env** .

The "centralized control" approach

This approach is applicable when the SystemVerilog and ***e*** parts need to interact during the test. In this case, the test implemented in ***e*** controls both ***e*** and SystemVerilog components.

You can view examples in:

- e Units Configuring SystemVerilog Components

6) Run the tests

For information about building the environment and running the tests, see Compiling and Running Multi-Language Environments

7) End the test

The methodology for ending the test ensures that:

1. All components are done with their test scenario.
2. The test is stopped gracefully; all components get to execute their post run phases.

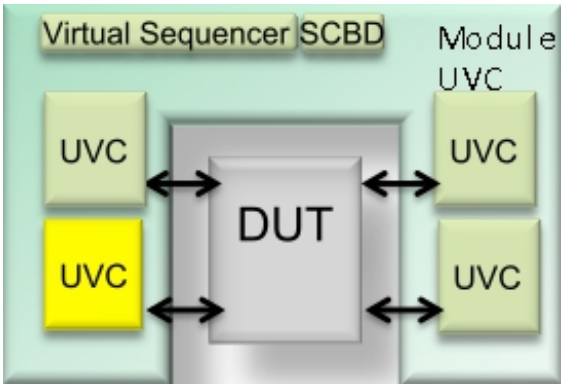
See Ending (Stopping) The Test

Integrating an e Unit into a SystemVerilog Verification Environment

This section describes how to integrate an ***e*** unit component into a SystemVerilog verification environment.

This section assumes that you are already familiar with the main principles of creating a system-level environment, all of which also apply to creating multi-language environments. This section focuses instead on the integration of UVCs implemented in different languages.

Following is an illustration of the basic model described in this section.



Following are the basic steps for integrating an ***e*** unit component into a SystemVeirlog verification environment. Each step is described in detail in the sections that follow in this guide:

1) Build the environment hierarchy

You can mix verification environments developed in different languages with UVM-ML. The components can be instantiated in a unified hierarchy or in a side-by-side hierarchy (parallel trees).

- Unified hierarchy--each component is instantiated in its logical location in the hierarchy.
  - For example, in an environment similar to the one shown in the picture above, four sub UVCs--three implemented in SystemVerilog and one implemented in ***e*** --would be instantiated in one module env.
  - For details, see Instantiating an e Unit Within a SystemVerilog Component .
- Side-by-side hierarchy (parallel trees)--the environment contains multiple tops, each top containing the components of one language.
  - For example, in an environment similar to the one shown in the picture above, there would be two tops. The SystemVerilog UVM top would contain the module env (which would contain three sub UVCs), and another top, **sys** , would contain the ***e*** sub UVC.

- For details, see Side-by-Side Multiple Tops Environment .

> For more information about unified hierarchies and side-by-side hierarchies, including illustrative figures, see "UVM-ML Hierarchy" in the "UVM-ML Key Concepts" chapter of the *UVM-ML OA Reference*

2) Configure the sub UVC

In a unified hierarchy, you can configure the sub-UVCs from the top environment. You might, for example, configure the number of agents to generate or the UVC work mode in this way. (Typical configuration tasks are described in SystemVerilog Components Configuring e Units .) When you configure an ***e*** unit in this way, it requires extending the ***e*** unit to get the configuration using **uvm_config_get()** .

(Typical configuration tasks are described in SystemVerilog Components Configuring e Units .)

3) Connect monitors

If any of the data collected by the ***e*** UVC is to be used in SystemVerilog components, for example in a system level checker, you need to instantiate ports in the checker and connect them to the ***e*** monitor ports. See e Units Passing Data to SystemVerilog Components .

4) Add multi-language sequences

Add multi-language sequences as per your test requirements. For example, if you want a virtual sequence that is implemented in SystemVerilog to **do** sequences and sequence items implemented in the e UVC, you could use multi-language sequences. See SystemVerilog Sequence do'ing e Sequences and Items .

5) Write tests

In UVM-SV, each test, implemented in a class extending **uvm_test** , configures the verification environment per test definition. This is done mainly with **uvm_config** . A typical UVM-SV test contains configuration settings that mainly direct the generation of the data items and choose the default sequence.

When the environment contains components in ***e*** , there are two main approaches to creating the UVM-SV tests:

- Independent behavior

- Central control

The "independent behavior" approach

In this approach, the behavior of the ***e*** components is not controlled from SystemVerilog components. This approach is applicable when the activities performed by the ***e*** components are independent of the activities performed by SystemVerilog components during the test. For example, the bus UVC is a reactive agent, responding to requests coming from the DUT.

If this matches your testing requirements, all you have to do is create an ***e*** test file controlling the ***e*** UVC and compile it with the rest of the environment. Note that each run (that is, simulation) should have only one **uvm test,** and in a SystemVerilog-over- ***e*** architecture it is a SystemVerilog **uvm_test** class.
Thus, even though there is an ***e*** test file, do not declare it as a **uvm test** .

The "central control" aproach

This approach is applicable when a test implemented in SystemVerilog controls both SystemVerilog and ***e*** components.

You can view examples in:

- SystemVerilog Components Configuring e Units and SystemVerilog Sequence do'ing e Sequences and Items .

6) Run the tests

For information about building the environment and running the tests, see Compiling and Running Multi-Language Environments .

7) End the test

The methodology for ending the test ensures that:

1. The test ends only after all components are done with their test scenario
2. The test is stopped gracefully; all components execute their post run phases

See Ending (Stopping) The Test .

See also:

- For more information about creating system verification environment, you can read "Module-to-System Verification" in the Cadence document *UVM **e** User Guide* .

Integrating a Reference Model Implemented in SystemC into an e Verification Environment

If you have a reference model implemented in SystemC, you will have to compare the results of the reference model to the results monitored from the DUT. In this case, you should implement ports passing items from the SystemC reference model to the ***e*** checker.

See:

- SystemC Components Passing Data to e Units

Integrating a Reference Model Implemented in SystemC into a SystemVerilog Verification Environment

If you have a reference model implemented in SystemC, you will have to compare the results of the reference model to the results monitored from the DUT. In this case, you should implement ports passing items from the SystemC reference model to the SystemVerilog checker.

See:

- SystemC Components Passing Data to SystemVerilog Components

Connecting Monitors of Various Languages, to One Checker

When integrating several verification components to one environment, you are typically required to connect a new checker to an existing environment. For example, you might need to add a system level checker that will get transactions from several monitors.

TLM ports are the basic tool for passing items between components implemented in difference languages, and we recommend that each monitor write the data items it collects on a port.

The recommended type of port is TLM analysis because of its broadcasting capability. When using TLM analysis ports, you can connect one TLM analysis port (in ***e*** , an *interface_port of tlm_analysis*) to multiple implementations (in ***e*** , an *interface_imp of tlm_analysis* or *interface_export of tlm_analysis*).

Note that you can connect the same port to multiple implementations implemented in different languages. For example, the monitor's port can be connected to a module level checker implemented in SystemVerilog and also to a scoreboard implemented in ***e*** .

See Also

- SystemVerilog Components Passing Data to e Units
- e Unit Passing Data to SystemC Components
- e Units Passing Data to SystemVerilog Components
- SystemC Components Passing Data to e Units

3

Creating a Multi-Language Environment

Typically, a verification environment contains a system UVC that itself contains instances of sub-system UVCs, bus UVCs, and maybe also some checkers.

In a UVM-ML environment, you can construct an environment of components implemented in various verification languages--for example, a system UVC implemented in ***e*** reusing a bus UVCs implemented in SystemVerilog, containing a checker implemented in SystemC.

To instantiate a component implemented in another language, you instantiate a proxy to the component. Actions performed on the proxy component are transferred to the actual component.

Currently, unifiied hierarchies support the following capabilities:

- Configuration of sub components.
- Synchronization of phases, which guarantees a predefined order in creation and configuration of components.
- Visualisation of the multi-language hierarchical tree.
- Support for multi-language hierarchical names for child component and their ports. The name is formed by concatenating the parent's path with the child's relative name-- that is, “ *parent-name-in-language1.child-relative-port-name-in-language2* ”.

Creating a unified hierarchy is recommended, but not required. You can alternatively create a side-by-side architecture (an environment containing parallel trees). Many UVM-ML capabilities function in side-by-side architecture, including connecting TLM ports for passing items among components.

This section describes how to create both kinds of hierarchies. The first two sub-sections describe creating a unified hierarchy in which ***e*** is on top and a unified hierarchy when SystemVerilog is on top. the last sub-section describes creating a side-by-side hierarchy.

This section contains:

See also:

- Configuration in a Multi-Language Environment
- "UVM-ML Hierarchy" in the "UVM-ML Key Concepts" chapter in the *UVM-ML OA Reference*
- "UVM-ML Configuration in a Unified Hierarchy" in the "UVM-ML Key Concepts" chapter in the *UVM-ML OA Reference*
- "Synchronized Phasing in a UVM-ML Environment" in the "UVM-ML Key Concepts" chapter in the *UVM-ML OA Reference*

Instantiating a SystemVerilog Component Within an e Unit

To instantiate a SystemVerilog component within an ***e*** unit, you first instantiate in ***e*** a proxy unit and then associate the proxy unit with the SystemVerilog component by constraining its **type_name** field. This proxy is now a "link" to the SystemVerilog component.

We recommend creating a new class in SystemVerilog that extends the component you will be reusing. In this extended component, you can add multi-language related capabilities or adjustments .

The steps to instantiate a SystemVerilog component within an e unit:

1. SystemVerilog:
    a. Recommended: Define a new component by extending the SystemVerilog component you want to instantiate within ***e*** . Add multi-language related functions as appropriate.
2. ***e*** :
    a. Instantiate in the ***e*** unit a unit of type **child_component_proxy.**
        a. Constrain its **type_name** field to match the type name of the SystemVerilog component.
        b. Optional: Configure this sub components using **uvm_config_set().**

The logical name of the SystemVerilog child component is the **e_path** of the e **child_component_proxy** instance. This name should be used when connecting TLM ports.

Code example:

The following example shows an instantiation of the SystemVerilog component *ubus_env* within an ***e*** unit named *testbench* .

The UVM name (the logical name) of the *ml_ubus_env* is the concatenation of the path of the containing unit + the name of the field " *sys.testbench.ubus_env.uvc_to* p".

- SystemVerilog:
  - Import and include the UVM packages **uvm_pkg** and **uvm_ml** (the adapter package).
  - Define the multi-language extended component by extending *ubus_env* and naming it *ml_ubus_env* .
  - Add getting the configuration with **uvm_config** , if it does not already exist in the code.
- ***e*** :
  - Define a unit to serve as a wrapper to the SystemVerilog component and name it *ubus_env .*
    - Instantiate in it a **child_component_proxy** , naming it *uvc_top.*
    - Constrain the child component proxy **type_name** to be the SystemVerilog component *ml_ubus_env.*
  - Set the configuration for *uvc_top.*
  - Instantiate the wrapper unit *ubus_env* in the *testbench* unit.

This code is copied from the example in `$UVM_ML_HOME/ml/examples/use_cases/e_over_sv` .

SystemVerilog code: extending the reused component

```
import uvm_pkg::*;
`include "uvm_macros.svh"
import uvm_ml::*;
class ml_ubus_env extends ubus_env;
`uvm_component_utils(ml_ubus_env)
function void build_phase(uvm_phase phase);
// Get configuration from container
void'(uvm_config_int::get(this, "", "slave0_max", slave0_max));
void'(uvm_config_int::get(this, "", "slave1_max", slave1_max));

// Propagate configuration to sub components
set_slave_address_map("slaves[0]", 0, slave0_max);
set_slave_address_map("slaves[1]", slave0_max+1, slave1_max);
endfunction : build_phase
endclass : ml_ubus_env
```

e code: instantiating the SystemVerilog class in e unit

```
unit ubus_env {
// Foreign (SV) child
// The path to the SV component ml_ubus_env will be {me.e_path; ".uvc_top"}.
uvc_top: child_component_proxy is instance;
keep uvc_top.type_name== "SV:ml_ubus_env";
// Configure UBUS
keep uvm_config_set("*", "slave0_max", 16'h7fff); // slave 0 address space
keep uvm_config_set("*", "slave1_max", 16'hffff); // slave 1 address space

// For connecting to its ports -
connect_ports() is also {
env_sequencer_0.connect_proxy(append(e_path(),
".uvc_top.masters[0].sequencer.seqr_tlm_if"));
env_sequencer_1.connect_proxy(append(e_path(),
".uvc_top.masters[1].sequencer.seqr_tlm_if"));
};
};

unit testbench like uvm_env {
ubus_env : ubus_env is instance;
};
```

See Also:

- e Units Configuring SystemVerilog Components
- "UVM-ML SystemVerilog Adapter Basics" in the *UVM-ML OA Reference*
- "UVM-ML e Instantiation of Foreign Child Components" in the *UVM-ML OA Reference*

Instantiating an e Unit Within a SystemVerilog Component

To reuse a component implemented in ***e*** within a SystemVerilog component, you must instantiate a component within the SystemVerilog component that will serve as a proxy to the ***e*** unit. To do this, you use **uvm_ml::uvm_ml_create_component** .

The steps to instantiate an e unit within a SystemVerilog component:

1. ***e*** :
    a. Extend the unit you want to instantiate to add multi-language capabilities.
        a. G et configuration, using **uvm_config_get(),** for all fields that are expected to be configured from the parent component.
        b. If the unit (or any unit under it) has to access the DUT (that is, if it contains a BFM or a monitor):
            a. Make sure you configure its **hdl_path()** and **agent().**
            b. Define a stub unit. See Stub Generation for an e Sub-Tree .
2. SystemVerilog:
    a. Instantiate a **uvm_component** under the appropriate component, that is, the `uvm_env` class. This component will be a proxy to the ***e*** unit.
    b. Create this child component using **uvm_ml_create_component().**

The logical name of the ***e*** unit will be the path of the SystemVerilog proxy class. This name is useful when connecting TLM ports.

Code example:

The following code example shows the steps for instantiating an ***e*** unit named x *bus_env_u* within the SystemVerilog class named *testbench* .

- ***e*** :
    - Extend *xbus_env_u* , the unit to be used within SV
        - Get configuration using **uvm_config_get** .
        - Note: if the unit is **like uvm_agent** or **uvm_env** , getting this configuration is already implemented, and thus there is no need to add it.
- SystemVerilog:
    - Implement the `env` component, naming it *testbench.*
        - Instantiate in it a **uvm_component** , naming it *xbus_uvc.*
        - Create the *xbus_uvc* field using **uvm_ml_create_component()** .

The code is copied form the example `$UVM_ML_HOME/ml/examples/use_cases/sv_over_e` .

e: extending a unit, adding ML functionality

```
       // If the unit does not derive uvm_agent or uvm_env, get the configuration of the
hdl_path() and agent()
       extend MY_XBUS xbus_env_u {
       keep uvm_config_get("agent()");
       keep uvm_config_get("hdl_path()");
       };
```

SystemVerilog code: instantiating the proxy, configuring the sub component

```
       class testbench extends uvm_env;
       uvm_component xbus_uvc;
       `uvm_component_utils(testbench)
       function void build_phase(uvm_phase phase);
       super.build_phase(phase);
       `uvm_info(get_type_name(), "testbench::build", UVM_LOW);
       xbus_uvc = uvm_ml_create_component("e", "xbus_env_u", "xbus_uvc", this);
       endfunction
       endclass : testbench
```

See also:

- Stub Generation for an e Sub-Tree
- SystemVerilog Components Configuring e Units
- "Specman Stub Generation for Sub-Trees in a Unified Hierarchy" in the *UVM-ML OA Reference*
- "UVM-ML SystemVerilog Instantiation of Foreign Child Components" in the *UVM-ML OA Reference*

Creating a Side-by-Side (Multiple Tops) Environment

Creating a unified hierarchy is recommended, but not required. You can alternatively create a side-by-side architecture, which is an environment containing multiple tops. All SystemVerilog components are instantiated under some **UVM** component, and all ***e*** units are instantiated under **sys** .

This architecture is useful when limited synchronization or cooperation is required between the various components. Examples of scenarios in which a side-by-side architecture is appropriate:

- A reference model implemented in SystemC passes data items to a scoreboard implemented in ***e*** .
- Two bus UVCs, each exercising and monitoring independently one of the DUT's interfaces, pass data items to a system-level scoreboard.

Limitations to a side-by-side architecture:

- Lack of multi-language configuration.
- The behavior of phase execution. Each phase is first executed on all components in one tree, that is, the phase tasks/methods of all the components under one top are called. Only then are the phase tasks/methods of the components under the next top executed.
    - You can manage this limitation by alternatively controlling the order of the phases with the SystemVerilog task **uvm_ml_run_test()** or the - **uvmtop** /- **uvmtest** arguments to irun. See details in " Synchronized Phasing in a UVM-ML Environment" in the "UVM-ML Key Concepts" chapter in the *UVM-ML OA Reference* .

Defining tops and the test from the command line:

Using IES, you can define the top/s and the test declaratively from the command line using the **irun** - **uvmtest** and - **uvmtop** flags.

If the test is in SystemVerilog, you can define it declaratively from the command line using the **+UVM_TESTNAME** flag.

Code example: defining multiple tops

The following SystemVerilog code defines the environment as having three tops:

- The ***e*** top file is named *top.e*
  - Provided to **irun**
- The SystemC top component is named *sctop.*
  - Passed to **uvm_ml_run_test()**
- The SystemVerilog top component is named *my_test* and is a **uvm_test** component.
  - Passed to **uvm_ml_run_test()**

SV code: Defining multiple tops

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

irun: Declaring three tops

```
irun top.e svtop.sv -sysc sctop.cpp -f ….
```

See also:

- Compiling and Running Multi-Language Environments
- "UVM-ML SystemVerilog Start of Test" in *UVM-ML OA Reference*
- "irun -uvmtest -uvmtop" in *UVM-ML OA Reference*

4

Type Mapping

In UVM-ML, data is passed between components using TLM ports or UVM configuration. The process of passing items requires:

- Knowing which type on the target side matches the type in the initiating language. This is referred to as type mapping.
- The serialization and de-serialization needs to match in order to get the correct data on the receiving side.

For ***e*** structs, the default is that physical fields (marked with **%**) are packed/unpacked.

For SystemVerilog items, the default is serialization of the fields that were added with the **uvm_field_** * macro.

For passing items between components, you should:

- Implement the data items:
  - Make sure both ports, in both languages, handle same data items.
    - If the items differ:
      - Implement a new type, in either language.
      - You might have to implement a converter, converting from the original data item handled by the UVC, to the new data item
- Instantiate and connect the required ports

In This Section

This section introduces some of the basics regarding type mapping and serialization, in the following sections:

See Also

For more complete information about type mapping and serialization, first read:

- "Data Communication in a UVM-ML Environment" in the "UVM-ML Key Concepts" chapter in the *UVM-ML OA Reference*

For information about the language constructs used to code type mapping and serialization in a UVM-ML environment, see:

- "UVM-ML SystemVerilog Serialization and Type Mapping" in the *UVM-ML OA Reference*
- "UVM-ML SystemC Serialization" in the *UVM-ML OA Reference*
- "UVM-ML e Type Mapping" in the *UVM-ML OA Reference*

Introduction to Type Mapping

Type-mapping refers to the process of mapping a composite type definition, such as a class or a struct, in one language to a corresponding type definition in another language. The consumer adapter needs to know to which type the producer type maps.

By default, the adapters look for matching types by name. If, for example, a struct named *xyz_transfer* is passed on a port, then the consumer adapter looks for a type named *xyz_transfer* .

In the following code blocks the two ports--in SystemVerilog and in ***e*** --pass matching items:

e code: data item and a TLM port

```
struct packet {
%data : int;
};
unit producer {
put_port : out interface_port of tlm_blocking_put of packet is instance;
```

SystemVerilog code: data item and a TLM export

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

class consumer#(type T=packet) extends uvm_component;
uvm_blocking_put_imp  #(T, consumer #(T)) put_export;
```

In situations in which the two type names differ, you can override the default name-matching rule with the following functions/statements:

- SystemVerilog: UVM ML adapter package function **set_type_match** (string *type-framework1* , string *type-framework2*)
- ***e*** : the statement **uvm_ml_type_match** *type-framework1 type-framework2*
- SystemC: No such function is urrently available. Instead, put the type match in the other language (call **set_type_match()** or **uvm_ml_type_match**)

The SystemVerilog and ***e*** functions must be called before any data communication, including configuration with objects.

The following example show how you specify that an ***e*** struct named *e_packet* matches the SystemVerilog class named *sv_packet:*

e code: override default mapping

```
struct e_packet {
%data : int;
};
uvm_ml_type_match "e:main:e_packet" "sv:my_pkg:sv_packet"
```

The following example shows the same functionality for SystemVerilog:

SystemVerilog code: overriding default mapping

```
 function void build_phase(uvm_phase phase);
int ml_res;
super.build_phase(phase);
ml_res = set_type_match("e:main:e_packet","sv:my_pkg:sv_packet"
);
```

Introduction to Serialization

Runtime serialization is used to convert transactions into a bit stream on the producer side, and de-serializarion is used to convert the list into a matching type in the consumer side.

For the consumer to get the right data after de-serialization:

- The bit sizes of the transactions must match
- The order of fields must be consistent

Existing default serialization:

- SystemVerilog: Using the **`uvm_field** _* macros automatically defines serialization (packing/unpacking).
- SystemC: there is predefined serialization of **tlm_generic_payload** and **tlm_extension_base** . For user defined types, you need to define the serialization explicitly.
- ***e*** : All physical fields (%) are automatically serialized.
  You can override the default serialization for e code to control which fields are to be de/serialized by extending the **tlm_pass_field** () method of **uvm_ml_config.**

Code example: SystemC manual serialization

The following code shows manual serialization of a SystemC class.

SystemC code: manual serialization

```
class packet : public uvm_object {
public :
UVM_OBJECT_UTILS(packet)
packet() { }
virtual ~packet() {}

virtual void do_pack(uvm_packer& p) const {
p << data;
// pack each element of array of objects
for (int i = 0; i < 4; i++) {
p << f_array_obj[i];
}
}
virtual void do_unpack(uvm_packer& p) const {
p >> data;
// unpack each element of array of objects
for (int i = 0; i < 4; i++) {
p >> f_array_obj[i];
}
}
public:
int data;
protected:
child_packet f_array_obj[4]; // array of child objects
};
UVM_OBJECT_REGISTER(packet)
```

Code example: e uvm_ml_config tlm_pass_field

The following code shows how you can define which fields are to be packed/unpacked when read/writing on ports.

e code: controlling which fields to be passed via TLM ports

```
struct ubus_transfer like any_sequence_item {
%addr : uint(bits:16);
%read_write : ubus_read_write_enum;
%data : list of byte;
%wait_state : list of uint(bits:4);
!%master : string;
!%slave : string;
};
extend uvm_ml_config {
tlm_pass_field(f:rf_field) : bool is also {
var s : rf_struct = f.get_declaring_struct();
var fname : string = f.get_name();
// Check for struct ubus_pkg::ubus_transfer
if ( s == rf_manager.get_type_by_name("ubus_pkg::ubus_transfer") ) {
// Pass only these fields:
if ( fname == "addr" ) { return TRUE; };
if ( fname == "read_write" ) { return TRUE; };
if ( fname == "data" ) { return TRUE; };
// All other fields - do not pass
return FALSE;
};
};
};
```

Using mltypemap

For irun users, you can use the **mltypemap** , which automates the definition of the target type and generates source the serialization code.

5

Configuration in a Multi-Language Environment

This section discusses pre-run configuration: setting verification environment parameters before the run phase.

- Some of the configuration is defined once per project, for example setting the number of PASSIVE masters and slaves in the DUT, as this depends on the DUT architecture.
- Other configuration aspects can vary over tests, for example setting the number of ACTIVE masters and slaves in the verification environment. These agents drive the stimuli to the DUT, and a different configuration might be required in different tests.

In a unified hierarchy (SystemVerilog, ***e,*** and SystemC components under one root) , you can propagate pre-run configuration values from one framework tree down to all sub-trees in other frameworks. To do so, you use the **native UVM configuration constructs** in each framework, for example, `uvm_config_db#(T)::set` in SystemVerilog and constraints with `uvm_config_set()` in ***e*** .

Each of the frameworks, UVM-SV, UVM-SC, and ***e,*** maintains a configuration database. When setting a configuration, the information is propagated and kept in each of the databases. When getting a configuration, the local framework gets the information from its database.

This section describes configuration in a unified hierarchy. It contains the following sub-sections:

- e Units Configuring SystemVerilog Components
- SystemVerilog Components Configuring e Units

See Also

- "UVM-ML Configuration in a Unified Hierarchy" in the "UVM-ML Key Concepts" chapter in the *UVM-ML OA Reference*

e Units Configuring SystemVerilog Components

If an ***e*** unit contains SystemVerilog components, their configuration can be set using **uvm_config.**

For configuration of SystemVerilog components from within *e* code:

1. SystemVerilog code:
    a. For each field that might be configured from above:
        a. If the **uvm_field_*** macro is not applied to this field, explicitly call **uvm_config_*()** during the creation of the component (for example, in **build_phase**), before this component is created.
        b. When passing a configuration object, use **uvm_config_object::get()** .

2. ***e*** code:
    a. For each SystemVerilog field you want to configure:
        a. Add a constraint using **keep uvm_config_set().**

Notes

- **uvm_config_*::get** must be called for the configuration to happen. It is automatically called for all fields specified with **uvm_field_** * macros.
- If there is a typo in the component or field name, there will be no warning message and this configuration will be ignored.
- When passing configuration object, make sure objects in both languages match. See Type Mapping .

See Also

- "UVM-ML SystemVerilog Configuration" and "UVM-ML e Configuration" in *UVM-ML OA Reference*

Code example 1: e configuring SystemVerilog

The following code shows the configuration of a SystemVerilog component from its containing ***e*** unit.

- SystemVerilog code:
  - In the **end_of_elaboration_phase** , get the configuration of the *slave0_max* and *slave1_max* fields.
- ***e*** code:
  - Using constraints, configure *slave0_max* to 16'h7fff, and *slave1_max* to 16'hffff.

The code shown below is a small part of the environment. You can see the full code in $UVM_ML_HOME/`ml/examples/use_cases/e_over_sv`.

SystemVerilog code: getting configuration from e

```
class ml_ubus_env extends ubus_env;
int slave0_max;
int slave1_max;
function void end_of_elaboration_phase(uvm_phase phase);
// Set the slave address space
void'(uvm_config_int::get(this, "", "slave0_max", slave0_max));
void'(uvm_config_int::get(this, "", "slave1_max", slave1_max));
set_slave_address_map("slaves[0]", 0, slave0_max);
set_slave_address_map("slaves[1]", slave0_max+1, slave1_max);
endfunction : end_of_elaboration_phase
endclass
```

e code: configuring SystemVerilog fields

```
extend testbench {
// configure UBUS.
// The following will configure all components that their full path contains the letters
"ubus_env".
keep uvm_config_set("*ubus_env*", "slave0_max", 16'h7fff); // slave 0 address space
keep uvm_config_set("*ubus_env*", "slave1_max", 16'hffff); // slave 1 address space
};
```

Code example 2: *e* configuring SystemVerilog using configuration object

The following code shows using **uvm_config_object** . For passing configuration objects, you define identical objects in ***e*** and SystemVerilog.

- SystemVerilog code:
  - Define the configuration object named *data* .
  - In the **build_phase** of the `env` component, get the configuration object.

- ***e*** code:
  - Define the configuration struct named *data* .
  - Using constraints, configure the data field of the SystemVerilog `env` component.

The code shown below is a small part of the environment. You can see the full code in $UVM_ML_HOME/`ml/examples/features/configuration/sv_e/sv_get_config` .

SystemVerilog code: getting coniguration object

```
class data extends uvm_transaction;
int    addr;
int    trailer;
string txt;
`uvm_object_utils_begin(data)
`uvm_field_int(addr, UVM_ALL_ON)
`uvm_field_int(trailer, UVM_ALL_ON)
`uvm_field_string(txt, UVM_ALL_ON)
`uvm_object_utils_end
endclass

class test extends uvm_env;
env sv_env;
uvm_object tmp_obj;
data conf_data;
pdata conf_pdata;
function void build_phase(uvm_phase phase);
super.build_phase(phase);
$display("SV test::build %s", get_full_name());
void'(uvm_config_object::get(this,"","conf_data", tmp_obj));
assert($cast(conf_data,tmp_obj)!=0);
// Can now configure the environment, using the fields in the configuration class
// for example -
set_config_int("*env","base_addr", conf_data.addr);
sv_env = new("sv_env", this);
endfunction
endclass
```

e code: setting configuration object

```
struct data {
% addr:int;
% trailer:int;
% txt:string;
};

unit u {
my_sv_child: any_unit is instance;
keep type my_sv_child is a child_component_proxy;
keep my_sv_child.type_name == "SV:test";
d : data;
keep d.addr == 10;
keep d.trailer == 20;
keep d.txt == "config object msg";

// Set the configuration object
keep uvm_config_set("my_sv_child","conf_data",d);
};
```

For more details, also read the following section:

- e Configuring SystemVerilog: Topology Configuration describes configuring the number of agents and their types (master/slaves, passive/active).

e Configuring SystemVerilog: Topology Configuration

In ***e*** verification environments, the environment is generated by the constraints solver, so topology is configured with constraints such as " *keep masters.size() == 3* ". In SystemVerilog, the environment is created procedurally. Each component configures and creates its sub components in **build_phase().**

In multi language environments, the ***e*** units can configure their sub components using **keep uvm_config_set()** . The SystemVerilog sub component should explicitly get the configuration by calling **uvm_config_*::get()** , unless the field was specified with the **uvm_field_*** macro which

automates the configuration.

For the *e* unit to configure the topology of the SystemVerilog component:

- The SystemVerilog component should contain fields used as directives for the generation. (*num_masters* , in the following example).
- The ***e*** code should use **uvm_config_set()** for passing the required value for the topology directives.
- The SystemVerilog component should get the values using **uvm_config** _* and continue creating the sub-components according to the values it got.

Code example:

The following code example adds configuration settings to the environment defined in Instantiating a SystemVerilog Component Within an e Unit . In this code, the top level unit, named *testbench* , configures the field *num_masters* of the *ubus_env* SystemVerilog component within it.

- ***e*** :
  - Extend the unit *testbench* , the unit containing the SystemVerilog component
    - Use **uvm_config_set()** - setting the field *num_masters* to 2, and *num_slaves* to 2.
- SystemVerilog:
  - The *ubus_env* gets the configuration and then creates the components under it.
    - This is done in the **build** phase
  - This code is part of the original UVC, implemented in `$UVM_ML_HOME/ml/examples/ex_single_lang_uvcs_lib/ubus_sv` .

This code is copied from the example `$UVM_ML_HOME/ml/examples/use_cases/e_over_sv`.

Code example: e unit configuring SystemVerilog components topology

```
extend testbench {
// configure UBUS
// configure all fields named num_masters, under components named ubus_top
keep uvm_config_set("*ubus_top*", "num_masters", 2);
keep uvm_config_set("*ubus_top*", "num_slaves", 2);
};
```

Code example: Topology configuration in the SystemVerilog code

```
class ubus_env extends uvm_env;
protected int num_masters = 0;
function void build_phase(uvm_phase phase);
// …
// Get required number of master agents
void'(uvm_config_db#(int)::get(this, "", "num_masters", num_masters));
// Create the master agents
masters = new[num_masters];
// Create masters, one by one
for(int i = 0; i < num_masters; i++) begin
masters[i] = ubus_master_agent::type_id::
create(inst_name, this);
// …
end
// …
endfunction : build_phase
endclass : ubus_env
```

See Also

- "UVM-ML SystemVerilog Configuration" and "UVM-ML e Configuration" in *UVM-ML OA Reference*

SystemVerilog Components Configuring e Units

If a SystemVerilog component contains ***e*** units, their configuration can be set using **uvm_config** .

For configuration of an e unit from within SystemVerilog code:

1. ***e*** code:
    a. For each field that might be controlled from SystemVerilog:
        a. Add a **keep soft uvm_config_get()** , if does not already exist.
2. SystemVerilog code:
    a. For each ***e*** field you want to configure:

a. Add **uvm_config_<>::set** prior to creating the component.

Notes

- You must call **uvm_config_get()** for the configuration to happen.
- If there is a typo in the unit or field name, there will be no warning message,and configuration will be ignored.

Code example: SystemVerilog configuring e

The following example is of the SystemVerilog component configuring the fields *min_addr* and *max_addr* of the ***e*** unit.

e code: getting configuration

```
unit xbus_config_u {
min_addr : xbus_addr_t;
max_addr : xbus_addr_t;
keep soft uvm_config_get(min_addr);
keep soft uvm_config_get(max_addr);
};
```

SystemVerilog code: configuring the e unit

```
class sys_env extends uvm_env;
xbus_proxy_t xbus_e_uvc;
uvm_config_int::set(this,"*xbus_config_u*","min_addr", 10);
uvm_config_int::set(this,"*xbus_config_u*","max_addr", 100);
xbus_e_uvc = xbus_proxy_t::type_id::create("xbus_e_uvc", this);
assert (xbus_e_uvc.create_foreign_component("e", "xbus_env_u") == 1);
endclass
```

In This Section:

See also:

- Instantiating an e Unit Within a SystemVerilog Component
- e Units Passing Data to SystemVerilog Components
- "UVM-ML SystemVerilog Configuration" and "UVM-ML e Configuration" in *UVM-ML OA Reference*

SystemVerilog Configuring e: Topology Configuration

In ***e*** , the verification environment is generated by the constraints solver. Thus, topology is configured with constraints such as " *keep masters.size() == 3* ". In SystemVerilog, the environment is created procedurally. Each component configures and creates its sub-components in **build_phase().**

In multi-language environments, the SystemVerilog components can configure their sub-components using **uvm_config_<>::set()** . The ***e*** sub-component should explicitly get the configuration by calling **uvm_config_get()** .

"Topology configuration" of a UVC means controlling the number of types of generated agents--for example, the number ACTIVE / PASSIVE agents of each kind (master, slave, arbiter).

For configuration of the e UVC from SystemVerilog:

1. The ***e*** unit should contain configuration knobs, f or example, *number_of_masters* , *number_of active_slaves* , ...
    a. Consider implementing a configuration struct that contains all configuration knobs.
2. The ***e*** unit should get the values of these knobs using **uvm_config_get()** .
3. The SystemVerilog component should call **uvm_config_<>::set()** to configure the ***e*** unit, prior creating it.

In some bus UVCs, the topology configuration is set by constraining the names of required agents (for example, in the sample XBUS UVC):

e code: configuration by setting names to the agents

```
extend MY_SYSTEM xbus_env_u {
keep passive_master_names == {};
keep active_master_names    == {MASTER_0; MASTER_1};
keep passive_slave_names  == {};
keep active_slave_names     == {SLAVE_0; SLAVE_1};
};
```

For such environments, it is recommend that a configuration struct be defined, containing a list of strings holding the names of agents to be generated. This configuration struct can be created in the SystemVerilog component and passed to the ***e*** environment using **set_config_object().**

When using configuration structs, make sure that the structs in both languages are aligned. See Type Mapping .

Code example:

The following code shows a SystemVerilog component setting the configuration of the XBUS sample UVC.

1. ***e*** :
    a. Implement a new configuration struct containing four lists of strings: one list per each agent kind (ACTIVE/PASSIVE MASTER/SLAVE)
    b. Get this struct using **uvm_config_get().**
    c. Constrain the list of the names based on the values in the configuration struct.
        a. Use **as_a()** to convert from strings to enums.
2. SystemVerilog:
    a. Configure the instantiated ***e*** unit to be of sub-type *XBUS_IN_ML.*
    b. Implement a new class, matching the ***e*** configuration struct.
    c. Register the configuration object to the database using **UVM_ML_CONFIG_DB_IMP** .
    d. Create and assign this class.
        a. In this example, a list of two names of active masters and two names of active slaves, is created. Thus the environment will contain four active agents.
    e. Pass this configuration class to the ***e*** unit using **uvm_config_object::set().**

```
struct xbus_topology_config {
%passive_master_names : list of string;
%active_master_names  : list of string;
```

```
    %passive_slave_names  : list of string;
    %active_slave_names   : list of string;
    };

    extend xbus_env_u {
    keep uvm_config_get(bus_name);
    };
    extend XBUS_IN_ML xbus_env_u {
    xbus_topology_config : xbus_topology_config;
    keep uvm_config_get(xbus_topology_config);
    // Constrain the names. The names are list of enums,
    // so converting from list of string, to list of the enum xbus_agent_name_t
    keep passive_master_names == xbus_topology_config.passive_master_names.as_a(list of
xbus_agent_name_t);
    keep active_master_names  == xbus_topology_config.active_master_names.as_a(list of
xbus_agent_name_t);
    keep passive_slave_names  == xbus_topology_config.passive_slave_names.as_a(list of
xbus_agent_name_t);
    keep active_slave_names   == xbus_topology_config.active_slave_names.as_a(list of
xbus_agent_name_t);
    };
```

SystemVerilog code: topology configuration of e

```
      extend xbus_bus_name_t : [XBUS_IN_ML = 1, FAST_MODE = 2, BMASTER = 3];
      class xbus_topology_config extends uvm_object;
      rand string passive_master_names[];
      rand string active_master_names[];
      rand string passive_slave_names[];
      rand string active_slave_names[];
      endclass : xbus_topology_config

      class sys_env extends uvm_env;
      // configuration of the xbus topology
      xbus::xbus_topology_config xbus_topology_config;
      // the proxy
      xbus_proxy_t xbus_e_uvc;
      function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      `UVM_ML_CONFIG_DB_IMP(  xbus::xbus_topology_config)
      // create topology configuration
      xbus_topology_config = new("xbus_topology_config");
      xbus_topology_config.active_master_names =  new[2];
      xbus_topology_config.active_slave_names = new[2];
      xbus_topology_config.active_master_names[0] = $psprintf("MASTER_0");
      xbus_topology_config.active_master_names[1] = $psprintf("MASTER_1");
      xbus_topology_config.active_slave_names[0]  = $psprintf("SLAVE_0");
      xbus_topology_config.active_slave_names[1]  = $psprintf("SLAVE_1");
      uvm_config_db#( xbus::xbus_topology_config)::set(this, "*xbus*","xbus_topology_config",
xbus_topology_config);
      // Configure the sub-type of the xbus env
      uvm_config_int::set(this,"*xbus_uvc","bus_name", XBUS_IN_ML);
      xbus_e_uvc = xbus_proxy_t::type_id::create("xbus_e_uvc", this);
      assert (xbus_e_uvc.create_foreign_component("e", "xbus_env_u") == 1);
      endfunction
      endclass : sys_env
```

See also:

- Instantiating an e Unit Within a SystemVerilog Component
- "The 'UVM_ML_CONFIG_DB_IMP' Macro" in the *UVM ML OA Reference*

Connecting the e Bus UVC to the DUT

One of the configuration tasks you must perform when integrating a new verification environment is connecting the bus UVC to the DUT.

The ***e*** UVC contains **simple ports** connected to the DUT, typically in the signal_map or synchroniser units (see "Basic UVC Architecture" in the Cadence manual *UVM e User Guide*).

You can configure the ***e*** units **hdl_path()** attribute from the SystemVerilog-containing component, using **uvm_config_** *.

In the configuration file:

1. If required, extend all relevant units: the top unit, the signal map, the synchronizer:
    a. You might have to constrain the **hdl_path()** of the unit.
    b. You might have to constrain the **hdl_path()** of the ports within this unit.

Code example

This code shows configuration of the synchronizer unit and the ports in it. The **hdl_path** of the top unit, *xbus_env_u* , is set from SystemVerilog.

- ***e*** code:
    - Extend the unit, adding getting the configuration with **uvm_config_get()** .
- SystemVerilog code:
    - Call **uvm_config_string::set()** , passing the path to the DUT

Note : This code is a small part of the environment, demonstrating just the configuraiton of the hdl_path. See the full code example in $UVM_ML_HOME/`ml/examples/use_cases/sv_over_e` .

e code: hdl_path and ports configuration

```
// Ensure the right subtype of the xbus_env_u is instantiated in this environment
extend xbus_env_u {
keep kind == XBUS_IN_ML;
};

extend XBUS_IN_ML xbus_env_u {
keep agent() == "SV";
};

extend XBUS_IN_ML xbus_synchronizer_u {
sig_clock_path : string;
keep uvm_config_get(sig_clock_path);
keep sig_clock.hdl_path() == value(sig_clock_path);
};
```

SystemVerilog code: configuration of hdl_path of the e unit

```
class svtop extends uvm_test;
uvm_config_string::set(this,"*xbus_uvc","hdl_path()", "~/xbus_evc_demo");
uvm_config_string::set(this,"*xbus_uvc*","sig_clock_path", "new_clock");
testbench tb

// Creating the SV component that contains the e xbus_uvc
tb = testbench::type_id::create("tb", this);
endclass
```

See also:

- Stub Generation for an e Sub-Tree

Stub Generation for an e Sub-Tree

Some ***e*** ports require generation of auxiliary code in a file commonly known as the "stub file". When the stub file is generated (for example, by **irun**), the unified multi-language testbench hierarchy is not yet known. As a result, when an ***e*** unit is to be instantiated within a SystemVerilog or SystemC component, the **write stub** is not aware of this unit, and this unit's ports are not taken into account during stub generation.

If you have such an ***e sub-tree*** , you should add the following to your code:

1. Define a parent proxy unit--in other words, a unit like **uvm_ml::parent_component_proxy** .
    a. Instantiate in it the ***e*** unit that is to be instantiated in a SystemVerilog or SystemC component.
    b. Configure the following attributes of the parent proxy unit to be exactly as expected to be in the unified hierarchy:
        a. `hdl_path()` -- The path to the relevant DUT module
        b. `agent()` -- SV
2. Extend **sys** :
    a. Extend **pre_stub_generation** ():
        a. Call **uvm_ml_create_stub_unit** ().
            a. The 1st parameter: the name (**- type**) of the parent proxy
            b. The 2nd parameter: the expected full path of the ***e*** sub-tree in the unified hierarchy

Code example:

The following code creates a stub unit for this environment:

- The top component is implemented in SystemVerilog and is named *uvm_test_top* . It contains:
    - A SystemVerilog component named *system_env,* containing:
        - An ***e*** unit named *xbus_env_u*

The code:

- SystemVerilog:
    - Create the ***e*** sub-tree, **create_foreign_component()** , and configure its **agent()** and **hdl_path().**
        - Set the **hdl_path()** to the relevant block in the DUT, the block to be accessed by *xbus_env_u - xbus_evc_demo* in this example.
        - See Instantiating an e Unit Within a SystemVerilog Component for more details.
- ***e*** :
    - Define a unit named *xbus_env_u,* the unit to be instantiated within the SystemVerilog component.

    - Get its hdl_path() using **uvm_config_get().**
- Define a unit like **parent_component_proxy** , naming it *stub_unit_for_xbus.*
    - Instantiate *xbus_env_u.*
    - Constrain its **hdl_path()** to the expected path of the relevant block in the DUT (*xbus_evc_demo* in this example).
    - Constrain its agent to "SV".
- Call **uvm_ml_create_stub_unit** (), in **sys.pre_stub_generation()** , specifying the expected path of the SystemVerilog container, *uvm_test_top.system_env.*

SystemVerilog code: creating e sub-tree in a SV component

```
// The proxy to the e unit
class xbus_proxy_t extends uvm_ml::child_component_proxy;
// adding fields and tasks ...
endclass

// Instantiating the proxy in the SV env component
class system_env extends uvm_env;
xbus_proxy_t xbus_uvc;

function void build_phase(uvm_phase phase);
super.build_phase(phase);
xbus_uvc = xbus_proxy_t::type_id::create("xbus_evc", this);
assert (xbus_uvc.create_foreign_component("e", "xbus_env_u") == 1);
endfunction
endclass

class svtop extends uvm_test;
system_env system_env;

function void build_phase(uvm_phase phase);
super.build_phase(phase);
// The path to the DUT
uvm_config_string::set(this,"*xbus_env*","hdl_path()","xbus_evc_demo");
uvm_config_string::set(this,"system_env","agent()","SV");
system_env = system_env::type_id::create("system_env", this);
endfunction
endclass
```

e code: defining a stub unit

```
unit xbus_env_u {
};

unit stub_unit_for_xbus like uvm_ml::parent_component_proxy {
xbus_env: xbus_env_u is instance;
// same path as expected to be set by the SV container
keep hdl_path() == "xbus_evc_demo";
keep agent() == "SV";
};

extend sys {
pre_stub_generation() is also {
uvm_ml_create_stub_unit("stub_unit_for_xbus","uvm_test_top.system_env.xbus_evc");
};
};
```

See also:

- "Specman Stub Generation for Sub-Trees in a Unified Hierarchy" in the *UVM-ML OA Reference*
- Instantiating an e Unit Within a SystemVerilog Component

6

Multi-Language Data Communication

UVM-ML uses TLM communication to manage communication between framework components. TLM communication is widely understood (the TLM standard) and is supported in all major verification languages. Plus, TLM communication encapsulates information as transactions, a concept that aligns well with object-oriented programming (transactions can be modeled as data objects).

Both TLM1 and TLM2 are supported in UVM-ML, and you can choose the port most applicable to your requirements.

For monitors, we recommend the **TLM1 analysis port** for its main advantage, namely the broadcast capability (the one-to-many connection). Using TLM1 analysis ports, one bus monitor can simultaneously provide data to its driver, a bus level checker, a system level checker, and more.

UVM-ML TLM communication is achieved by serializing the transaction content, sending it to the connected port, and de-serializing it there. This process requires three things:

- UVM-ML needs to know which type to map to which type (type mapping).
- The serialization and de-serialization needs to match, in order to get the same transaction on each side.
- The order of fields in composite types (such as classes or structs) must be consistent.

For more details about UVM-ML requirements for TLM communication, see "Data Communication" and "UVM-ML TLM Communication" in the "UVM-ML Key Concepts" chapter of the *UVM-ML OA Reference.*

In This Section:

See also:

- Type Mapping

SystemVerilog Components Passing Data to e Units

This section describes the use model of a SystemVerilog component that needs to pass data to a component implemented in ***e*** , for example, a monitor passing items to a system level checker. It is assumed that the SystemVerilog monitor already writes the data item on an analysis port and that now you need to connect this port to a port in the ***e*** unit.

The steps for doing so are as follows:

1. Make sure both ports, in both languages, handle matching data types.
    a. Create data item definitions, mapping the ***e*** item to a SystemVerilog one, or vice versa. See Type Mapping .
2. Implement the ***e*** port:
    a. Extend the ***e*** monitor/checker.
        a. Instantiate the in analysis port.
        b. Implement the port **write()** method.
3. Register the ports:
    a. SystemVerilog: register the port at the end of the build phase using **uvm_ml::ml_tlm1::register().**
    b. ***e*** : bind the port to external using a **keep bind(** *port* **, *external)*** constraint.
4. Connect the ports:
    a. Should be done once, either from SystemVerilog code or from **e** code.
        a. Connecting ports in SystemVerilog:
            a. In **connect_phase(),** connect the ports using **uvm_ml::connect().**
        b. Connecting ports in e:
            a. In **connect_ports()** , connect the ports using **uvm_ml.connect_names().**

When instantiating an in port in ***e*** , we recommend defining the port using a prefix or suffix, to prevent naming collision in case you instantiate in this unit additional ports in the future. For

example:

```
   ubus_transfer_ip : in interface_port of tlm_analysis of ubus_transfer using
prefix=ubus_ is instance;
```

The **write** method of this port, to be named "ubus_write()", should implement the required activity of the system monitor/checker when getting a transfer from the ubus. See the example below.

Note:

Connecting the ports is done by logical names (strings). There is no checking of type matching, or even whether the ports exist. To get the path to a port:

- You can use, for example, the SystemVerilog functions **print_topology()** or **get_full_name()** , and the ***e*** method **e_path().**
- You can also use the interactive command **uvm_ml trace_register_tlm** . It is recommended to apply this command once you have the registration code (**ml_tlm1#(T)::register(), ml_tlm2#(T1,T2)::register()**) in place and before adding **uvm_ml.connect_names()** in ***e*** code **.**
- You can also use commands such as **uvm_component -list** or **uvm_ml print_tree.**

Code example:

The example below is of connecting a port in a monitor implemented in SystemVerilog to a system monitor implemented in ***e*** .

The structure of the environment:

- sys (***e*** top unit)
    - *xcore_sve* (***e*** unit)
        - *system_monitor* (***e*** unit)
        - *ubus_env* (SystemVerilog component, represented by *ubus_env_proxy* (***e*** unit, proxy to SystemVerilog))
            - *bus_monitor* (SystemVerilog component)
                - *item_collected_port* (uvm_analysis_port#(ubus_transfer))

The logical name (full path) of the SystemVerilog port is *sys.xcore_sve.ubus_env_proxy.bus_monitor.item_collected_port.*

The steps for passing data from the SystemVerilog *bus_monitor* to the ***e*** *system_monitor* :

1. SystemVerilog:
    a. Define the port.
        a. The port is defined in the reused UVC, UBUS, in the *ubus_bus_monitor* component.
    b. Edit the extended component, the component defined for the multi-language environment as described in Instantiating a SystemVerilog Component Within an e Unit .
        a. Register the *bus_monitor* port.
2. ***e*** :
    a. Define a struct, identical to the *ubus_transfer* class defined in the UBUS SystemVerilog UVC.
    b. Extend *system_monitor* , adding an in **analysis TLM implementation (in interface_port of tlm_analysis)**
        a. Implement the port, processing the item as required (for example, checking its fields).
    c. Bind the new port to external.
    d. Connect the new port to the port in the monitor.
        a. Use **connect_names.** The path to the port is *sys.xcore_sve.ubus_env_proxy.bus_monitor.item_collected_port* .

SystemVerilog code: port definition

```
class ubus_bus_monitor extends uvm_monitor;
// Analysis ports for the item_collected and state notifier.
uvm_analysis_port #(ubus_transfer) item_collected_port;

// ...
endclass: ubus_bus_monitor
```

SystemVerilog code: register the port to uvm_ml

```
class ml_ubus_env extends ubus_env;
function void phase_ended(uvm_phase phase);
// Register ML ports
if (phase.get_name() == "build") begin
uvm_ml::ml_tlm1 #(ubus_transfer)::register(bus_monitor.item_collected_port);
end
endfunction
endclass : ml_ubus_env
```

e Code: connect the ports

```
// This struct matches the SystemVerilog ubus_transfer class
struct ubus_transfer like any_sequence_item {
%addr : uint(bits:16);
%read_write : ubus_read_write_enum;
%size : uint;
%data : list of byte;
%wait_state : list of uint(bits:4);
!%master : string;
!%slave : string;
}; // end struct ubus_transfer
extend system_monitor {
// Adding the analysis TLM implementation
//
ubus_transfer_ip : in interface_port of tlm_analysis of ubus_transfer
using prefix=ubus_ is instance;

// Implementing the port
//
ubus_write(transfer : sv_ubus_transfer) is {
// Process the item, eg - check its fields
check SYSTEM_LEGAL_ADDRESS that transfer.addr >=  me.base_address;
// ...
};
connect_ports() is also {
do_bind(ubus_transfer_ip, external);

// Connection is done using the logical name.
// We get the names using e_path()
var path_to_sv_port : string =
append(get_enclosing_unit(xcore_sve_u).e_path(),
".ubus_env_proxy.bus_monitor.item_collected_port");
var connected :=
uvm_ml.connect_names(path_to_sv_port,
ubus_transfer_ip.e_path());
check that connected else
dut_error("Ports connection failed for ports ",
past_to_sv_port, " and ",
ubus_transfer_ip.e_path() );
};
};


unit xcore_sve_u like uvm_env {
system_monitor : system_monitor is instance;
ubus_env_proxy: child_component_proxy is instance;
keep ubus_env_proxy.type_name== "SV:ubus_env";
};
```

See Also:

- Using mltypemap, described in Type Mapping .
- "UVM-ML SystemVerilog TLM Interface" in the *UVM-ML OA Reference*
- "UVM-ML e TLM Interface" in the *UVM-ML OA Reference*
- "Incisive Debugging Commands for Use in a UVM-ML Environment" in the *UVM-ML OA Reference*

e Units Passing Data to SystemVerilog Components

This section describes the use model of an ***e*** component that needs to pass data to SystemVerilog components, for example, a monitor passing items to system level checker. We assume that this ***e*** monitor already uses a TLM port for passing data, and now you need to connect this port to a port in the SystemVerilog component.

The steps for doing so are as follows:

1. Make sure both ports, in both languages, handle compatible data items.
    a. Create data item definitions, mapping the **e** item to a SystemVerilog one, or vice versa. See Type Mapping .
2. Implement the SystemVerilog port:
    a. Implement the port either in the original definition of the component or in a new class extending the original monitor/checker:
        a. Instantiate the port implementation
        b. Implement the **write()** function, implementing the required behavior when the analysis imp receives the incoming data.
3. Register the ports:
    a. SystemVerilog: register the port at the end of the build phase, using **uvm_ml::ml_tlm1::register()** .
    b. ***e*** : bind the port to external, using a **keep bind(** *port* **, external)** constraint.
4. Connect the ports:
    a. Should be done once, either from SystemVerilog code or from ***e*** code.
        a. Connecting ports in SystemVerilog:

   a. In **connect_phase(),** connect the ports using **uvm_ml::connect()** .
b. Connecting ports in e:
   a. In **connect_ports()** , connect the ports using **uvm_ml.connect_names()** .

Example:

The following example demonstrates connecting a port implemented in an ***e*** unit to a port in a system-level monitor, implemented in SystemVerilog.

The top level of the ***e*** UVC is of type *xbus_env_u* . We instantiate it in a SystemVerilog component named *sys_env,* which is instantiated under the test. Consequently, the environment hierarchy is:

- *uvm_test_top*
  - *env* (SystemVerilog component)
    - *xbus_trans_ended_imp* (SystemVerilog implementation port)
    - *xbus_e_uvc* (the name of the child_component_proxy instance)
      - *bus_monitor* (***e*** unit)
        - *transfer_ended_o* (the port)

The full name of the port is *uvm_test_top.env.xbus_e_uvc.bus_monitor.transfer_ended_o.*

For passing data from the ***e*** bus_monitor to the SystemVerilog env component, we:

1. Define an *xbus_trans_s* type in SystemVerilog.
2. ***e*** :
   a. Extend the port container, which is the *bus_monitor.*
      a. Bind the port to external. This port has already been defined in this unit, in the UVC we reuse (the XBus sample UVC).
3. SystemVerilog:
   a. Instantiate a TLM analysis implementation and implement its **write()** function.
   b. Register the port, in **phase_ended()** .
   c. Connect the ports, in **connect_phase()** .

The code examples below show only the parts of code relevant for creating and connecting ports.

e code: binding the port to external

```
extend xbus_bus_monitor_u {
keep bind(transfer_ended_o, external);
};
```

SystemVerilog code: instantiate the e unit and the port, and connect the port

```
class sys_env extends uvm_env;
uvm_ml::child_component_proxy xbus_e_uvc;
// getting data from bus monitor
uvm_analysis_imp#(xbus_trans_s, sys_env) xbus_trans_ended_imp;
`uvm_component_utils(sys_env)
function new(string name, uvm_component parent=null);
super.new(name,parent);
// ...
xbus_trans_ended_imp = new("xbus_trans_ended_imp", this);
endfunction

function void build_phase(uvm_phase phase);
super.build_phase(phase);
// ...
xbus_e_uvc = xbus_proxy_t::type_id::create("xbus_e_uvc", this);
assert (xbus_e_uvc.create_foreign_component("e", "xbus_env_u") == 1);
};
function void phase_ended(uvm_phase phase);
// Register ML ports
if (phase.get_name() == "build") begin
uvm_ml::ml_tlm1 #(xbus_trans_s)::register(xbus_trans_ended_imp);
end
endfunction
function void connect_phase(uvm_phase phase);
super.connect_phase(phase);
if(!uvm_ml::connect( {get_full_name(), ".xbus_e_uvc.bus_monitor.transfer_ended_o"},
xbus_trans_ended_imp.get_full_name() )) begin
`uvm_fatal("CON_TO_XBUS", "uvm_ml connect failed");
end;
endfunction : connect_phase
// write - what to do when writing on the port
virtual function void write(xbus_trans_s trans);
//.. check the data ....
endfunction : write
endclass : sys_env

// The top, test component
class svtop extends uvm_test;
sys_env env;
endclass : svtop
```

See Also:

- Using mltypemap, described in Type Mapping .
- "UVM-ML SystemVerilog TLM Interface" in the *UVM-ML OA Reference*
- "UVM-ML e TLM Interface" in the *UVM-ML OA Reference*

SystemVerilog Components Passing Data to SystemC Components

This section describes the use model of a SystemVerilog component that needs to pass data to a component implemented in SystemC; for example, a monitor passing items to a reference model.

The content of this section will be added soon.

Code examples exist in `$UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv` .

See Also

- "UVM-ML SystemC TLM Interface" in the *UVM-ML OA Reference*

SystemC Components Passing Data to e Units

This section describes the use model of a SystemVerilog component that needs to pass data to a component implemented in SystemC; for example, a reference model passing items to a checker.

The content of this section will be added soon.

Code examples exist in `$UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv_e` .

See Also

- "UVM-ML SystemC TLM Interface" in the *UVM-ML OA Reference*

SystemC Components Passing Data to SystemVerilog Components

This section describes the use model of a SystemC component that needs to pass data to a component implemented in SystemVerilog; for example, a reference model passing items to a checker.

The content of this section will be added soon.

Code example exists in `$UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv`.

See Also

- "UVM-ML SystemC TLM Interface" in the *UVM-ML OA Reference*

e Unit Passing Data to SystemC Components

This section describes the use model of an ***e*** component that needs to pass data to SystemC components, for example, a monitor passing items to a checker. We assume that this ***e*** monitor already uses a TLM port for passing data, and now you need to connect this port to a port in the SystemC component.

The best mechanism for passing items is using ports, and the recommended port type is ***TLM analysis*** port, which allows broadcasting.

To pass items from a monitor implemented in ***e*** to a checker implemented in SystemC:

1. Make sure both ports, in both languages, handle compatible data items.
    a. Create data item definitions, mapping the ***e*** item to a SystemC one, or vice versa. See Type Mapping.
2. Register the ports:
    a. SystemC: Register the port using **uvm_ml_register()**.
    b. ***e***: bind the port to external using a **keep bind(** *port* **, external)** constraint.

3. Connect the ports:
    a. Should be done once, either from SystemC code, or from **e** code.
        a. Connecting the ports in SystemC:
            a. In **before_end_of_elaboration()** , connect the ports using **uvm_ml_connect()** .
        b. Connecting the ports in ***e*** :
            a. In **connect_ports()** , connect the ports using **uvm_ml.connect_names()** .

Code example: connecting an e monitor port to a SC checker port

The following code demonstrates passing a struct from ***e*** to SystemC. It contains these parts:

1. SystemC:
    a. Port instantiation, registration and implementation
2. ***e*** :
    a. Port instantiation and connection
    b. Writing on port

SystemC code: port instantiation, registration and implementation

```
class checker : public sc_module, tlm_analysis_port<my_xbus::xbus_trans_s> {
public:
checker(sc_module_name nm) : sc_module(nm), get_trans("get_trans") {
get_trans(*this);
}
sc_export<tlm_analysis_if<my_xbus::xbus_trans_s> > get_trans;
void write(const my_xbus::xbus_trans_s value) {
// do something with this item... check it...
};
};
class sctop : public uvm_component    {
public:
checker sc_checker;
sctop(sc_module_name nm) : uvm_component(nm), sc_checker("sc_checker") {
uvm_ml_register(&sc_checker.get_trans);
}
```

e code: port instantiation and connection

```
extend system_monitor {
// ports for items matching the item used in the SC checker
trans_ended_op : out interface_port of tlm_analysis of common_xbus_trans_s is instance;
keep bind(trans_ended_op, external);
connect_ports() is also {
compute uvm_ml.connect_names(trans_ended_op.e_path(),
"sctop.sc_checker.get_trans");
};
};
```

e code: data conversion & writing on port

```
extend system_monitor {
// type matching the type used in the SC checker
cur_transfer :  common_xbus_trans_s;
event trans_ended;
on trans_ended {
// Write on port - to be passed to the checker
trans_ended_op$.write(cur_transfer);
};
};
```

See Also

- "UVM-ML SystemC TLM Interface" in the *UVM-ML OA Reference*
- "UVM-ML e TLM Interface" in the *UVM-ML OA Reference*

7

Multi-Language Sequence Layering

There are several use models for using a sequence library implemented in another language:

- The simplest use model, requiring minimal interaction between the components, is that the top level controls which sequence kind should be started when the test begins. This means configuring the sequencer's default sequence.
- An alternative use model involves continuous interaction between sequences implemented in different languages. In this use model, a virtual sequence implemented at the system level is ***do'*** ing sequences of the sub UVCs.
    - This can be implemented in phases. First, have the top level sequence **do** 'ing items of the low level sequence. Then, add the ability for it to **do** sequences.

In both of these use models, the lower level should provide API support to the upper level. That means that you cannot just take ***e*** code and do its sequence from SystemVerilog code, or vice versa. Some extensions are required.

This chapter describes the steps required for implementing these use models for the following architectures: ***e*** component controlling SystemVerilog sequences and SystemVerilog component controlling ***e*** sequences.

This chapter contains:

e Component Constraining the Default SystemVerilog Sequence

When writing a test using a sequence library implemented in UVM-SystemVerilog, the simplest way to define the test scenario is to configure the default sequence, and thus determine which of the sequences in the sequence library will be started when the run begins. This is applicable also in a multi-language environment; the ***e*** components can configure the default sequence using the UVM-ML configuration API.

Following is a typical implementation in SystemVerilog UVCs to allow upper layers to choose the default sequence:

- SystemVerilog:
  - Define a field of string to hold the name of the default sequence (slave_seq_name in the following code example).
  - Get the value of this field using **uvm_config_string::get().**
  - Find a sequence matching the required name, using the UVM factory.
  - Configure **default_sequence,** using **uvm_config_db** , to the sequence type you got in the previous step.

For the e code to configure the **default_sequence** :

- In ***e*** code:
  - Constrain the name of the SystemVerilog default sequence using **keep uvm_config_set**

Code example:

The folowing code is taken from the sequence layering examples in `$UVM_ML_HOME/ml/examples/use_cases/e_over_sv`.

SystemVerilog getting default sequence from upper layer

```
      class ml_ubus_env extends ubus_env;
      string slave_seq_name;
      function void build_phase(uvm_phase phase);
      uvm_factory factory = uvm_factory::get();
      uvm_object_wrapper seq_wrap;
      super.build_phase(phase);
      // Select the slave sequences
      void'(uvm_config_string::get(this, "", "slave_seq_name", slave_seq_name));
      seq_wrap = factory.find_by_name(slave_seq_name);
      if(seq_wrap == null) begin
      uvm_report_warning("", $sformatf("Could not find sequence by the name %s",
slave_seq_name));
      end else begin
      uvm_config_db#(uvm_object_wrapper)::set(this,
      "slaves[0].sequencer.run_phase",
      "default_sequence",
      seq_wrap);
      end
      endfunction
      endclass : ml_ubus_env
```

e Constraining the SystemVerilog Default Sequence

```
      extend testbench {
      // config the field named slave_seq_name, of the component that is instantiated
      // in me.ubus_env.uvc_top
      keep uvm_config_set("ubus_env.uvc_top", "slave_seq_name", "slave_memory_seq");
      };
```

See also:

- "Setting and Retrieving Configuration Values in a UVM-ML Unified Hierarchy" in the *UVM-ML OA Reference*
- "keep uvm_config_set()" in the *UVM-ML OA Reference*
- UVM SV sequences and usage of **default_sequence** in the Accelera *UVM Class Reference*
- For a more complex use model, with the sequences interacting during the run, see e Sequence do'ing SystemVerilog Sequences and Items

e Sequence do’ing SystemVerilog Sequences and Items

In a UVM-ML environment, ***e*** sequences can **do** sequences and items implemented in SystemVerilog. The way to do this is to:

1. Instantiate a sequencer proxy in the ***e*** code.
2. Instantiate a sequencer TLM Interface in the SystemVerilog code.
3. Connect the two.

You should also export the sequence item and the sequences.

In This Section

The following sub-sections describe the steps required for **do** 'ing SystemVerilog sequence items and sequences from ***e*** code:

Adding a TLM Interface to the SystemVerilog Sequencer

To enable **do** 'ing SystemVerilog sequences from ***e*** , add a TLM API to the the SystemVerilog sequencer, which can be done by instantiating in it an **ml_seqr_tlm_if** --a sequencer TLM Interface. This class is implemented in UVM-ML primitives and contains the TLM implementations required for connecting to ***e*** .

This interface will be connected to the ***e*** sequencer proxy, as shown in Creating an e Proxy Sequencer .

Following are the steps for adding the TLM interface:

1. Define a class extending the template class **ml_seqr_tlm_if.**

   a. When using this template, you have to use the relevant sequence item and sequence type as template parameters.
   b. If the sequence has to send a response to the ***e*** sequence, you should implement the **update_response** task.
2. Define a class extending the sequencer you want to export (ubus_master_sequencer in the following example). In it:
   a. Instantiate the **TLM interface** implemented in the previous step.
   b. Set the sequencer pointer of the **TLM interface** , calling its method **set_seqr** ().
3. In the env component instantiating the sequencer, use the UVM factory to override its type to use the new type (the one containing TLM Interface).

Code example

The following code is taken from the example in `$UVM_ML_HOME/ml/examples/use_cases/e_over_sv` .

The code shows:

- SystemVerilog:
  - Defining a TLM interface, named *ubus_master_seqr_tlm_if* , extending **ml_seqr_tlm_if** .
    - Using the sequence item and the sequence type, *ubus_transfer* and *ubus_base_sequence,* as the template types .
  - Defining a sequencer named *ubus_master_ml_sequencer* , extending the sequencer we want to access from ***e*** , ubus_master_sequence.
    - Instantiating in it a *ubus_master_seqr_tlm_if.*
    - Setting the sequencer pointer of the TLM Interface.
  - Overriding the sequencer type in its container.

SystemVerilog Code: Exporting SystemVerilog Sequencer

```
// Define the Sequencer TLM Interface
class ubus_master_seqr_tlm_if extends ml_seqr_tlm_if#(ubus_transfer,ubus_base_sequence);
//...
endclass : ubus_master_seqr_tlm_if
// Master sequencer with TLM IF replacing the original sequencer
class ubus_master_ml_sequencer extends ubus_master_sequencer;
ubus_master_seqr_tlm_if seqr_tlm_if;

//..
function void build_phase(uvm_phase phase);
super.build_phase(phase);
seqr_tlm_if = ubus_master_seqr_tlm_if::type_id::create("seqr_tlm_if", this);
endfunction
// Set the sequencer pointer of the TLM interface
function void end_of_elaboration_phase(uvm_phase phase);
void'(seqr_tlm_if.set_seqr(this));
endfunction
endclass : ubus_master_ml_sequencer
class ml_ubus_env extends ubus_env;
function void build_phase(uvm_phase phase);
super.build_phase(phase);
// substitute the sequencer to add the TLM interface
set_type_override_by_type(ubus_master_sequencer::get_type(),
ubus_master_ml_sequencer::get_type());
endfunction
endclass : ml_ubus_env
```

See also:

- Creating an e Proxy Sequencer

Creating an e Proxy Sequencer

To enable ***e*** code **do** ing items of a SystemVerilog sequence, define a proxy sequencer (sequence driver) in ***e*** . This sequencer is of items corresponding to the items of the target SystemVerilog sequence.

You have to define and implement these types:

1. A sequence item.

    a. Define a struct **like any_sequence_item** , corresponding the SystemVerilog UVC sequence item. This struct should contain physical fields that are identical to the fields of the SystemVerilog sequence item.

        a. You can create it using Type Mapping .

2. A sequence base.
    a. Define a struct like **any_sequence_item** , corresponding the SystemVerilog UVC sequence.
    b. For doing items (and not sequences), this struct can be defined as empty.
3. A proxy sequencer.
    a. Import **sequence_layering/e/ml_base_seqr_proxy** . This file is in `$UVM_ML_HOME/ml/primitives` , which is in the SPECMAN_PATH.
    b. Define a sequencer using the **sequence** macro.

        a. Its item is the item defined in step 1; its sequence_driver_type is **ml_base_seqr_proxy** of the sequence defined in step 2.
    c. Implement the **body()** of this sequence.

        a. It should call **driver.send_sequence()** , passing to it the input sequence struct.
    d. We recommend defining a method of *get_exported_seq().* The **body()** should call it, and it can be extended in the sequence sub types.

        a. See more in e Sequence do'ing SV Sequences .
    e. Connect this sequencer to the Sequencer TLM Interface defined in SystemVerilog (see Adding a TLM Interface to the SystemVerilog Sequencer), using the **connect_proxy()** method, which connects all the TLM ports at once.

After implementing these types, you can do the proxy items in the proxy sequencer, and they will be propagated to the SystemVerilog sequencer.

For example, following are the types defined in the ubus example and the types defined in ***e*** to be able to drive the SystemVerilog sequences:

Ubus SystemVerilog UVC contains:	***e* UVC driving the Ubus contains:**	
`class ubus_transfer extends uvm_sequence_item`	`struct ubus_transfer like any_sequence_item`	
`class ubus_base_sequence extends uvm_sequence #(ubus_transfer)`	`struct ubus_base_sequence like any_sequence_item { }`	

`class ubus_master_sequencer extends uvm_sequencer #(ubus_transfer)`	`sequence ubus_master_sequence using item = ubus_transfer, sequence_driver_type = ml_base_seqr_proxy of (ubus_base_sequence), created_driver = ubus_master_seqr_proxy;`

Code example: creating proxy items in *e*

The following code is copied from the example in `$UVM_ML_HOME/ml/examples/use_cases/e_over_sv`. It shows:

- SystemVerilog:
  - The sequence and sequence item we want to **do** from ***e*** . This piece of code is copied from the SystemVerilog UVC.
- ***e*** :
  - The sequence and sequence item corresponding to the sequence and sequence item SystemVerilog.
  - The proxy sequencer:
    - Defining the sequencer.
    - Instantiating it in the env unit.
    - Connecting it to the SystemVerilog sequencer.
  - Usage example - **do** 'ing a SystemVerilog item from ***e*** .

SystemVerilog code: the sequence item and the sequence to be reused in the ML environment

```
class ubus_transfer extends uvm_sequence_item;
rand bit [15:0]           addr;
rand ubus_read_write_enum read_write;
rand int unsigned         size;
rand bit [7:0]            data[];
rand bit [3:0]            wait_state[];
rand int unsigned         error_pos;
rand int unsigned         transmit_delay = 0;
string                    master = "";
string                    slave = "";
endclass : ubus_transfer

class ubus_base_sequence extends uvm_sequence #(ubus_transfer);
endclass : ubus_base_sequence
```

e code: sequence and sequence item

```
struct ubus_transfer like any_sequence_item {
%addr : uint(bits:16);
%read_write : ubus_read_write_enum;
%size : uint;
%data : list of byte;
%wait_state : list of uint(bits:4);
%error_pos : uint;
%transmit_delay : uint;
!%master : string;
!%slave : string;
};

// The base type; to be derived for each sequence type
struct ubus_base_sequence like any_sequence_item { };
```

e code: proxy sequencer

```
import sequence_layering/e/ml_base_seqr_proxy;
sequence ubus_master_sequence using
item = ubus_transfer,
sequence_driver_type = ml_base_seqr_proxy of (ubus_base_sequence),
created_driver = ubus_master_seqr_proxy;

extend ubus_master_sequence {
// Virtual methods, to be extended in the derived structs
get_exported_seq() : ubus_base_sequence is {
result = NULL;
};
body()@sys.any is only {
driver.send_sequence(get_exported_seq());
};
};

unit ubus_env_proxy {
// Instantiating the proxy defined by the sequence macro above
env_sequencer : ubus_master_seqr_proxy is instance;
// Foreign (SV) child, the env containing the sequencer
sv_ubus_env: child_component_proxy is instance;
keep sv_ubus_env.type_name== "SV:ml_ubus_env";
connect_ports() is also {
// Connect the ports in the proxy sequencer to the ports
// in the TLM interface of the SV sequencer
env_sequencer.connect_proxy(append(e_path(),
".sv_ubus_env.master.sequencer.seqr_tlm_if"));
};
};
```

See also:

- For the full syntax of the ml_base_seqr_proxy, see `$UVM_ML_HOME/ml/docs/html_docs/sequence_layering_e_proxy/index.html`.

e do'ing SystemVerilog Sequence Items

After you have added a TLM interface to the SystemVerilog sequencer (see Adding a TLM Interface to the SystemVerilog Sequencer) and an ***e*** proxy sequencer (see Creating an e Proxy Sequencer), you can **do** SystemVerilog sequence items from ***e*** sequences.

You **do** the proxy items in the ***e*** proxy sequencer, and they will be propagated to the SystemVerilog sequencer.

Code example: doing proxy items in *e*

The following code is based on the code example shown in Creating an e Proxy Sequencer , showing how you can **do** proxy sequence items. These items will be propagated to the SystemVerilog sequencer.

e code: doing e proxy items

```
extend ubus_master_sequence_kind : [WRITE_READ];
extend WRITE_READ ubus_master_sequence {
!req     : ubus_transfer;
body() @driver.clock is only {
var save_ad  : int;
driver.raise_objection(TEST_DONE);
message(LOW, "WRITE_READ sequence starting ");
// Write and read back item
// These action will automatically drive the items
// to the SystemVerilog sequencer
do req keeping {
.read_write == WRITE;
.addr in [0 .. 4095];
.size == 4;
};
message(LOW, "wrote item ", hex(req.data));
save_ad = req.addr;
do req keeping {
.read_write == READ;
.addr == save_ad;
.size == 4;
};
};
};
```

e do'ing SystemVerilog Sequences

After you have defined the proxy types in ***e*** (see Creating an e Proxy Sequencer), you can **do** SystemVerilog sequences from ***e*** . For each sequence kind in SystemVerilog you want to drive from ***e*** , you should:

- Export the SystemVerilog Sequence:
  - Define the serializer for it. When structs/class objects are passed via the ports, they are serialized. The serializer is responsible for translating the data type coming from ***e*** , unpacking the stream of bytes into the class fields.
- Define the ***e*** proxy sequence:
  - Add a new struct, derived from the base ***e*** proxy sequence (*ubus_base_sequence* , in our example).
    - Add fields matching the exported SystemVerilog sequence.
- Extend the ***e*** multi-language sequence:
  - Extend the enumeration type kind of the multi-language sequence kind (*ubus_master_sequence_kind* , in our example) and add the new kind.
  - Extend the new sequence subtype, implementing the functionality required for creating the required sequence.
    - Implement its **get_exported_sequence** (), to return the requested sequence.

Code example: exporting SystemVerilog sequences, *e* proxy sequences

The following code is taken from the example in `$UVM_ML_HOME/ml/examples/use_cases/e_over_sv` .

It shows the implementation of an ***e*** proxy to a SystemVerilog sequence named *write_word* .

SystemVerilog code: The exported SV sequence

```
class write_word_seq extends ubus_base_sequence;
rand bit [15:0] start_addr;
rand bit [7:0] data0; rand bit [7:0] data1;
rand bit [7:0] data2; rand bit [7:0] data3;
rand int unsigned transmit_del = 0;
endclass : write_word_seq

// Sequence serializer
class ubus_pkg_write_word_seq_serializer extends uvm_ml::uvm_ml_class_serializer;
function void deserialize(inout uvm_pkg::uvm_object obj);
ubus_pkg::write_word_seq inst;
int tmp_size; // used for deserialization of arrays
$cast(inst,obj);
inst.start_addr = unpack_field_int(16);
inst.data0 = unpack_field_int(8);
inst.data1 = unpack_field_int(8);
inst.data2 = unpack_field_int(8);
inst.data3 = unpack_field_int(8);
inst.transmit_del = unpack_field_int(32);
endfunction : deserialize
endclass : ubus_pkg_write_word_seq_serializer
```

e code: proxy sequences

```
// Fields required for driving the SV write_word sequence
struct write_word_seq like ubus_base_sequence {
%start_addr : uint(bits:16);
%data0 : byte;
%data1 : byte;
%data2 : byte;
%data3 : byte;
%transmit_del : uint;
};

// Extending the sequence proxy, implementing the WRITE_WORD proxy
extend ubus_master_sequence_kind : [WRITE_WORD];
extend WRITE_WORD ubus_master_sequence {
exported_seq : write_word_seq;
get_exported_seq() : ubus_base_sequence is {
result = exported_seq;
};
};
```

See also:

- e do'ing SystemVerilog sequence items

SystemVerilog Component Constraining the MAIN e Sequence

When writing a test using an ***e*** sequence library, the simplest way to define the test scenario is to constrain the MAIN sequence, thus determining which of the sequences in the sequence library will be started when the run begins. This is applicable also in a multi-language environment: Using the UVM configuration API, SystemVerilog components can configure the ***e*** sequencer. You should extend the ***e*** sequencer so that it calls **uvm_config_get()** as part of generating the MAIN sequence.

Constraining the MAIN sequence is done as follows:

1. in ***e*** code:
    a. Add to the ***e*** sequencer (sequence driver) a field of string, for example name it *default_sequence.*
    b. Add a constraint getting the value of the *default_sequence* field using **uvm_config_get().**
    c. The MAIN sequence should constrain its sub-sequence to the value of its **driver** . *default_sequence.*
    d. Optionally, you can do the same for the **count** field of MAIN sequence.
2. in SystemVerilog code:
    a. The SystemVerilog component containing the ***e*** sequencer should configure the field *default_sequence,* using **uvm _config_string:get().**

Code example: SystemVerilog configuring the *e* MAIN sequence

The following code example shows how SystemVerilog can configure which ***e*** sequence will run.

- ***e*** : the ***e*** MAIN sequence calls **uvm_config_get()** , getting the value for its sub-sequence kind and the count (number of times to execute this sequence).
- SystemVerilog: a call to **uvm_config_string:s:set()** configures the sequence kind, and a call

to **uvm_config_int::set()** configures the count.

the e code:

```
<'
extend ubus_sequencer_u {
default_sequence : string;
running_count    : uint;
// getting configuration from above
keep uvm_config_get(default_sequence);
keep uvm_config_get(running_count);
};

extend MAIN ubus_sequence {
// constraining based on parent sequencer
keep soft sequence.kind ==
driver.default_sequence.
as_a(ubus_sequence_kind);
keep soft count == driver.running_count;
};
'>
```

the SystemVerilog code:

```
class svtop extends uvm_test;
/// instantiating sub components
function void build_phase(uvm_phase phase);
// building …
// Configure all sub components having
// “driver” in their path to send 10
// sequences of type SEND_MANY
uvm_config_string::set(this, "*driver",
"default_seq", "SEND_MANY");
uvm_config_int::set(this, "*driver","count", 10);
// continue creating the environment, including the
// env that contains drivers
sv_env = xbus_env::type_id::create("sv_env", this);
endfunction
endclass
```

See also:

- For more complex use model, with on-the-run interaction between the sequences, see SystemVerilog Sequence do’ing e Sequences and Items

SystemVerilog Sequence do’ing e Sequences and Items

SystemVerilog sequences can **do** sequences and items implemented in ***e*** . The way to do this is to:

1. Instantiate a sequencer proxy in the SystemVerilog code.
2. Instantiate a sequencer TLM Interface in the ***e*** code.
3. Connect the two.

You should also export the sequence item and the sequences.

In This Section

The following sub-sections describe the steps required for **do** 'ing ***e*** sequences from SystemVerilog code:

Adding a TLM Interface to the e Sequencer

To enable driving ***e*** sequences and sequence items from SystemVerilog sequences, add a TLM API:

1. Import the file **$UVM_ML_HOME/ml/primitives/sequence_layering/e/seqr_tlm_interface,** containing the definition of the ml_seqr_tlm_if. **$UVM_ML_HOME/ml/primitives** is in SPECMAN_PATH.
2. Define an multi-language designated sequence. This new kind of sequence (named *ML_SERVICE_SEQ* in the example below) will handle **do** requests coming from

SystemVerilog.

   a. Define in it a method named **do_user_seq** (), as empty. The next sections show how to extend it.

3. Instantiate a multi-language sequencer **ml_seqr_tlm_if** in the environment (for example, in the sequencer you export).

   a. The sequencer should be **of** the sequence data item and the type you defined in the previous step.

      a. For example, *xbus_trans_s* and *ML_MASTER_SERVICE_SEQ xbus_master_sequence,* as in the following example.

Code example: adding a TLM interface to an *e* sequencer

The following code illustrates adding a TLM interface to a sequencer named *xbus_master_driver_u,* of sequence item *xbus_trans_s.* The code shows *:*

- Defining a new sequence kind: *ML_MASTER_SERVICE_SEQ* .
- Extending the new sequence sub type.
  - Defining **do_user_seq()** as empty
- Instantiating an **ml_seqr_tlm_if** in the sequencer (sequence driver). It should be of the sequence defined in the previous step.

e code: exporting a sequence driver

```
import sequence_layering/e/seqr_tlm_interface;
extend ML_MASTER_SERVICE_SEQ xbus_master_sequence {
do_user_seq(p : ml_seq) @ driver.clock is empty;
};
// Instantiate the sequencer tlm interface
extend xbus_master_driver_u {
tlm_if: ml_seqr_tlm_if of (xbus_trans_s,
ML_MASTER_SERVICE_SEQ xbus_master_sequence)
is instance;
keep tlm_if.sequencer == me;
};
```

See also:

- For the full syntax of the ml_seqr_tlm_if, see `$UVM_ML_HOME/ml/docs/html_docs/sequence_layering_e_tlm_if/index.html`.

Instantiate a SystemVerilog Sequencer Proxy

To **do** ***e*** sequences and items from SystemVerilog code, instantiate a sequencer proxy (**ml_sequencer_proxy**) within a proxy component (a **child_component_proxy**). This proxy component should derive **child_component_proxy** , which implements all required infrastructure required for running multi-language sequences. Now you can **do** ***e*** sequence items on this sequencer proxy.

You should:

- Implement and connect the sequencer proxy:
  - Implement a proxy component, deriving class **child_component_proxy** (*xbus_proxy_t* , in the following example)
    - In it, instantiate a sequencer proxy of type **ml_sequencer_proxy**
  - Instantiate the proxy component in your environment (for example, in the proxy class, as shown in Instantiating an e Unit Within a SystemVerilog Component)
  - Connect the sequencer proxy to the tlm_if in the related ***e*** ml sequencer, the one defined in Adding a TLM Interface to the e Sequencer .

Code example, implementing a sequencer proxy in SystemVerilog

The code below Illustrates creating the sequencer proxy. For a full example, see `$UVM_ML_HOME/ml/examples/use_cases/sv_over_e`.

SystemVerilog code: implementing the sequencer proxy

```
`include "sequencer_proxy.sv"
class xbus_proxy_t extends child_component_proxy;
ml_sequencer_proxy seqr_proxy_0;
function void build_phase(uvm_phase phase);
super.build_phase(phase);
seqr_proxy_0 = ml_sequencer_proxy::type_id::
create("seqr_proxy_0", this);
endfunction
`uvm_component_utils(xbus_proxy_t)
endclass
class xbus_env extends uvm_env;
xbus_proxy_t xbus_uvc;
function void build_phase(uvm_phase phase);
super.build_phase(phase);
// instantiate the e env unit
xbus_uvc = xbus_proxy_t::type_id::
create("xbus_evc", this);
assert(xbus_uvc.
create_foreign_component(“e”,
“xbus_env_u”)==1);
endfunction
function void connect_phase(uvm_phase phase);
xbus_proxy_t e_proxy;
super.connect_phase(phase);
xbus_uvc.seqr_proxy_0.connect_proxy(
{xbus_uvc.get_full_name(),
".active_masters[0].
ACTIVE'MASTER'driver.tlm_if."});
endfunction : connect_phase
endclass : xbus_env
```

See also:

- Adding a TLM Interface to the e Sequencer
- SV do'ing Exported e Sequences
- For the full syntax of the proxy sequencer, see `$UVM_ML_HOME/ml/docs/html_docs/sequence_layering_sv/files/sequencer_proxy-sv.html`.

SystemVerilog do'ing e Sequence Items

To **do** ***e*** sequence items from SystemVerilog code, do the following:

- ***e*** :
  - Implement an **ml_seqr_tlm_if** (the one you instantiate in your environment as shown in Adding a TLM Interface to the e Sequencer **).** In it:
    - Implement the **process_item()** method, creating an ***e*** item based on the item it got from SystemVerilog.
      - The **process_item()** method gets the item as passed from SystemVerilog and should return the sequence item to be passed to the ***e*** sequence driver. It can create an item identical to the one that came from SystemVerilog or modify some of its fields. For example, it might check for correctness of the data item it got and modify them accordingly.
    - Optionally: Implement the **restore_item()** method **.**
      - The **restore_item()** method passes a response to the calling sequence. Implementing this method is required when the proxy ***e*** sequencer has to return a response to the caller (for example, as a result of a read transaction).
- SystemVerilog:
  - Implement a sequence item, matching the e sequence item (see Type Mapping). (*xbus_trans_s* in the following example)
  - Implement a sequence, designated to **do** the ***e*** items, containing a field of the type defined in the previous step.
  - Connect the sequence to the proxy sequencer using **uvm_declare_p_sequencer.** For definition of the proxy sequencer, see Instantiate a SystemVerilog Sequencer Proxy

The items you do in this sequence will be transferred to the sequencer proxy, and from it to the ***e*** sequencer to which the sequencer proxy was connected.

Code example: SystemVerilog sequence doing items

The following code shows:

- e:
  - Definition of the sequence item *xbus_trans_s.*
  - Implementation of the **procss_item()** and **restore_item()** methods of **ml_seqr_tlm_if** .

- SystemVerilog:
  - Definition of the matching sequence item *xbus_trans_s.*
  - Definition of the sequence *xbus_seq.*
  - **do** 'ing the sequence item.

For a full example, see `$UVM_ML_HOME/ml/examples/use_cases/sv_over_e` .

e Code: implementing ml_seqr_tlm_if

```
struct xbus_trans_s like any_sequence_item {
kind : xbus_trans_kind_t;
%addr : xbus_addr_t;
%size_ctrl : uint (bits : 2);
%read_write : xbus_read_write_t;
size : uint [1, 2, 4, 8];
%data[size] : list of byte;
};
extend MASTER xbus_trans_s {
wait_states : list of uint (bits : 4);
error_pos_master : int;
transmit_delay : uint;
};

extend ml_seqr_tlm_if of (MASTER xbus_trans_s,
ML_MASTER_SERVICE_SEQ xbus_master_sequence) {
// generate a valid item from the SV parameters
process_item(p : any_sequence_item) :
MASTER xbus_trans_s is {
var inp := p.as_a(MASTER xbus_trans_s);
gen new_item keeping {
.driver           == sequencer;
.addr             == inp.addr;
.data             == {inp.data};
.size             == inp.size;
.read_write       == inp.read_write;
.size_ctrl        == inp.size_ctrl;
.wait_states      == {inp.wait_states};
.error_pos_master == inp.error_pos_master;
.transmit_delay   == inp.transmit_delay;
};
return new_item;
};

restore_item() : MASTER xbus_trans_s is {
result = save_item;
};
};
```

Code example: SystemVerilog sequence do'ing items of e sequence

```
class xbus_trans_s extends uvm_sequence_item;
rand xbus_trans_kind_t kind;
rand xbus_addr_t addr;
rand bit[1:0]  size_ctrl;
rand xbus_read_write_t read_write;
rand int unsigned size;
rand byte unsigned data[];
rand bit [3:0] MASTER__wait_states[];
rand int MASTER__error_pos_master;
rand int unsigned MASTER__transmit_delay;
// ...
endclass : xbus_trans_s
class xbus_seq extends uvm_sequence;
xbus_trans_s      xb_item;   // xbus sequence item
int               target_addr;
`uvm_declare_p_sequencer(ml_sequencer_proxy)
virtual task body();
// Write 8 random bytes
`uvm_do_with(xb_item, {addr == 'h1000;
read_write == WRITE;
size == 8;
})
// Write 4 bytes
target_addr = $urandom() & 'hffff;
`uvm_do_with(xb_item, {addr == target_addr;
read_write == WRITE;
size == 4;
foreach (data[i]) {i==0 -> data[i]=='hde;
i==1 -> data[i]=='had;
i==2 -> data[i]=='hbe;
i==3 -> data[i]=='hef;}
})
endtask // body
endclass : xbus_seq
```

See also:

- Instantiate a SystemVerilog Sequencer Proxy

SystemVerilog do'ing Exported e Sequences

For **do** 'ing ***e*** sequences from SystemVerilog, you have to implement a proxy--a mediator between the ***e*** sequence and a SystemVerilog sequence. ***e*** sequences are implemented as WHEN subtypes. To be able to invoke them from SystemVerilog, we must convert them to the equivalent "like" representation, which is compatible with the SystemVerilog sequence methodology.

For each ***e*** sequence you want to enable **do** 'ing from SystemVerilog, you should:

- ***e*** : Export the sequence:
  - Implement an **ml_seq** struct:
    - This struct will be passed on the port from SystemVerilog and, based on its fields, the ***e*** sequence will be created. In the example below, we name the struct *WRITE_TRANSFER* .
    - Extend the method **do_user_seq()** in the multi-language designated sequence, the sequence handling the **do** 's from SystemVerilog (See Adding a TLM Interface to the e Sequencer), adding the handling of the new exported sequence.
- SystemVerilog: Create a proxy sequence:
  - Define a class in SystemVerilog, deriving **ml_seq** , for each exported sequence. This class should:
    - Contain the same fields as the exported ***e*** sequence.
    - Use **uvm_declare_p_sequencer** .
  - Define a sequence, deriving **uvm_sequence** . In this sequence:
    - Define a field of the type of the exported sequence.
    - Use **uvm_declare_p_sequencer** .
    - In the sequence body(), you can **do** the exported sequence.

Code example: SystemVerilog sequence do'ing exported *e* sequences

The code below illustrates:

- Creating the sequence wrapper in ***e*** , defining a struct containing fields required for doing the xbus WRITE_TRANSFER sequence.
- The implementation of the **do_user_seq()** in the exported sequence.
- Doing this exported sequence in SystemVerilog code.

For a full example, see `$UVM_ML_HOME/ml/examples/use_cases/sv_over_e` .

e code: the ml_seq

```
struct WRITE_TRANSFER like ml_seq {
% base_addr : uint(bits:16);
% size      : uint;
% data      : uint(bits:64);
};
```

e code: implementing do_user_seq()

```
extend ML_MASTER_SERVICE_SEQ xbus_master_sequence {
!wt_user_seq : WRITE_TRANSFER xbus_master_sequence;
do_user_seq(p : ml_seq) @ driver.clock is also {
if(p.kind == "WRITE_TRANSFER") {
do wt_user_seq keeping {
.base_addr ==
p.as_a(WRITE_TRANSFER).base_addr;
.size == p.as_a(WRITE_TRANSFER).size;
.data == p.as_a(WRITE_TRANSFER).data;
};
};
};
};
```

SystemVerilog sequence do'ing the exported e sequence

```
class WRITE_TRANSFER extends ml_seq;
rand bit[15:0] base_addr;
rand int unsigned size;
rand bit[63:0] data;
`uvm_declare_p_sequencer(ml_sequencer_proxy)
function new(string name="WRITE_TRANSFER");
super.new(name);
kind = name;
endfunction
endclass : WRITE_TRANSFER

class xbus_seq extends uvm_sequence;
WRITE_TRANSFER    wt_seq;    // xbus WRITE_TRANSFER sequence
int               target_addr;
`uvm_object_utils(xbus_seq)
`uvm_declare_p_sequencer(ml_sequencer_proxy)
virtual task body();
// Invoke exported sequences
target_addr = $urandom() & 'hffff;
`uvm_do_with (wt_seq, {base_addr == target_addr;
size == 4;})
endtask
endclass : xbus_seq
```

See also:

- Instantiate a SystemVerilog Sequencer Proxy

8

Ending (Stopping) The Test

Currently, UVM-ML OA does not provide synchronization between frameworks for run-time phases. We recommend you implement an End of Test synchronisation, using ports.

Any methodology for ending the test should ensure that:

1. All components are done with their test scenario.
2. The test is stopped gracefully, and all components get to execute their post-run phases.

Both ***e*** and UVM-SV contain objection mechanisms that support these requirements. Each component should raise an objection when it starts its test activities and drop the objection when done. (In ***e*** , you should call `raise_objection(TEST_DONE)` , as there can be more kinds of objections.) The test stops only after all components are done with their test activities. After the test is stopped, the port-run phases (extract, check, finalize) are called.

Currently, there is no multi-language objection mechanism; Specman and UVM-SystemVerilog do not coordinate objections raises and drops. Until this capability is added to UVM-ML, you should implement synchronization between components implemented in different languages.

Regardless of the environment hierarchy, SystemVerilog-over- ***e*** or ***e*** -over-SystemVerilog, we recommend centralizing the end-of-test synchronization in one of the SystemVerilog components.

Note:

- In multi-language environments, do not call Specman's **stop_run().** When the simulation is stopped from this routine, the post-run phases implemented in SystemVerilog components are not called.

Implementing End-of-Test Synchronization

For implementing an End-of-Test synchronisation between SystemVerlog and ***e*** , you should implement a simple protocol between SystemVerilog and ***e*** components, for ***e*** to indicate that all its objections are dropped. You can use TLM ports for this, and also simple ports, as shown below.

1. The SystemVerilog component should raise objection when the test starts.

2. When in the ***e*** side all objections to `END_TEST_DONE` are dropped, the ***e*** unit should write on the port to indicate that its objection to End of Test is dropped.
3. When the SystemVerilog sees this indication, it can drop its objection.

Code example: synchronizing End of Test

In the following example, we define a **reg** in a SystemVerilog module and connect it to a **simple_port** in ***e*** .

The implementation:

1. **SystemVerilog** *code **:***
    a. Instantiate a reg, named *e_objections_dropped.*
    b. When the run starts and **run_phase()** is called; raise objection to end of phase.
    c. When the *e_objections_dropped* is set, drop the objection.
2. ***e*** code:
    a. Instantiate a **simple_port** and connect it to the *e_objections_dropped reg.*
    b. Extend **all_objections_dropped()** .
        a. When all objections to **TEST_DONE** are dropped, write the port.

SV code: raising/dropping objection, per indication from e

```
module topmodule;
// get from e indication of end of test
reg e_objections_dropped;
};

class ubus_example_tb extends uvm_test;
///...
task run_phase(uvm_phase phase);
phase.raise_objection(this);
wait_for_e_end_of_test();
endtask : run_phase
task wait_for_e_end_of_test();
@topmodule.e_objections_dropped;
`uvm_info(get_type_name(),
$sformatf("Got indication from e, can drop objection\n\n"),
UVM_LOW);
uvm_test_done.drop_objection(this);
endtask
endclass
```

e code: informing SystemVerilog of objections dropped

```
extend sys {
all_objections_dropped_op : out simple_port of bit is instance;
keep bind(all_objections_dropped_op, external);
keep all_objections_dropped_op.hdl_path() == "~/topmodule.e_objections_dropped";
all_objections_dropped(kind: objection_kind) is only {
// Do not stop the run. Instead -  inform SystemVerilog
if kind == TEST_DONE {
message(LOW, "Informing the objections to TEST_DONE dropped");
all_objections_dropped_op$ = 1;
};
};
};
```

9

Compiling and Running Multi-Language Environments

This chapter explains how to **compile** , **elaborate** , and **run** a UVM-ML environment.

Most of the information is provided in simulator specific sub-chapters.

This chapter contains:

Prerequisites for Running UVM-ML: A Checklist

To use UVM-ML, you need to install the package, include it in your flow, and the verification code has to import and utilize UVM-ML.

- **Installation** needs to be done once per release, see the installation guide within the release at `$UVM_ML_HOME/ml/README_INSTALLATION.txt` .
- **Setup** needs to be done once per session, see the `$UVM_ML_HOME/ml/README_INSTALLATION.txt` file and search for the word `setup` .
- **Code preparation starts** with importing the UVM-ML package, adapters and constructs, see the "Adapter Basics" sections for SystemVerilog and SystemC in the UVM-ML reference manual.
- **Code preparation continues** with utilizing the specific ML features you need, see Creating a Multi-Language Environment for details.

Once the above prerequisites are met, you can compile and run your environment and tests.

Using ‘demo.sh --dry-run’

This section shows how to run UVM-ML and how to learn about the various steps and options from the provided demos.

The examples under the directory `$UVM_ML_HOME/ml/examples/ use_cases/` focus on usage flows, and can be used to view and explore compilation options. All these examples have a `demo.sh` script to demonstrate the various run options.

We will use the **`/use_cases/side_by_side/sc_sv_e`** example for demonstration. To run this example, invoke the `demo.sh` script in one of two ways:

```
% $UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv_e/demo.sh <simulator>
or
% $UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv_e/demo.sh <simulator> --dry-run
```

The first option runs the whole example with the requested simulator, and you can observe the output.

The second option, running demo.sh with --dry-run, is a way to explore the build flow: it prints to the screen all the commands used to run this example, without running it. This provides visibility into the steps taken to build and run the example and shows all the command line options used.

> The information printed to the screen can be placed into a file and then run "as is", as a shell script. You can also edit this file and modify it to your needs.

To get a listing with explanations of all `demo.sh` options, use

```
% demo.sh -help
```

Examples of the output obtained from running `demo.sh <simulator> --dry-run` with the various simulators are provided in the following simulator specific sections.

See also:

- Running UVM-ML with Incisive (IES)
- Running UVM-ML with VCS
- Running UVM-ML with Questa

Running UVM-ML with Incisive (IES)

This section shows how to compile and run a UVM-ML environment with the Cadence Incisive Enterprise Simulator (IES). It describes both the irun (single step) flow and the multi-step flow.

Section Contents

Running an Example with demo.sh

This section shows the commands used for the /side_by_side/sc_sv_e/ example. The options used for a unified hierarchy example would be slightly different (for example, there would not be multiple -uvmtops options).

The output from the dry-run was shortened for clarity, as indicated by the "..." markers. To see the full printout you can run the command interactively.

Turning on the Incisive UVM-ML Mode

To run UVM-ML simulations with Incisive, you need to turn on the UVM-ML mode, in order to benefit from some special services: loading of the shared libraries (e.g., backplane, UVM-SC), activating the UVM-ML phasing, etc.

The UVM ML mode can be turned on by placing one or more of the the following options on the irun or ncsim invocation lines: `-ml_uvm` or `-uvmtop` or `-uvmtest`.

The `-ml_uvm` option turns on the UVM-ML mode explicitly. If either the `-uvmtop` or `-uvmtest` options are present, this turns on the UVM-ML mode implicitly. All combinations of using these 3 flags are supported.

You will not see an explicit use of `-ml_uvm` in the examples of this section, because it is already included in the `ml_options.[32|64].f` files.

Using the irun Compilation Flow (single-step)

Using **irun** combines the compile, elaborate and run steps into one invocation, thus reducing the number of steps the user needs to do.

```
% cd $UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv_e/
% demo.sh IES -dry
#!/bin/sh
# Dry-run output: the command(s) used to run this test are:
...
# ies_run_command :
irun ./test.sv -f ${UVM_ML_HOME}/ml/run_utils/ml_options.32.f \
-f ${UVM_ML_HOME}/ml/run-utils/sv_[<uvm-sv-version>_]options.32.f \
-f ${UVM_ML_HOME}/ml/run_utils/e_options.32.f \
-f ${UVM_ML_HOME}/ml/run_utils/sc_options.32.f \
-uvmtop SV:svtest -uvmtop e:./top.e -uvmtop SC:sctop \
./sc.cpp -l irun_ncsc_cl.32.log -exit
...
```

Inputs : the code files to compile or load:

- ./test.sv - a SystemVerilog file
- ./top.e - an ***e*** file
- ./sc.cpp - the SystemC file

Flags :

Flag	Description
`-f`	Specifies argument files, one generic and one for each language involved
`-uvmtop`	Specifies the top components for each hierarchy tree
`-l`	Provides a name for a log file
`-exit`	Indicate IES to exit when the simulation is completed

For information about the general options, see "irun Options Used with UVM-ML".
For information about the '-f' argument files see "The Provided irun Argument Files".

Using the Multi-Step Compilation Flow

If you are interested in running multi-step compilation, rather than single-step irun, then run one of the basic examples in $UVM_ML_HOME/ml/examples/use_cases/, and use the –steps option, as in:

```
% demo.sh IES -dry -steps ies_ncsc_proc_multistep
```

> The `-steps` option should be last, and accepts an optional <name-of-make-target> argument after it. You can run `demo.sh -help` for more details on the options available for multi-step.

Multi-step compilation (also known as 3-step) is described in detail in the section "Incisive Multi-Step Compile and Run Flow".

The Provided irun Argument Files

Running simulations with multiple languages requires many options to be provided to the simulators. For IES, these options were organized into several argument (.f) files:

- General ML options: ml_options.32.f , ml_options.64.f
- SystemVerilog options: sv_1.1_options.32.f, sv_1.1_options.64.f and the matching files for SystemVerilog 1.2: sv_1.2_options.32.f, sv_1.2_options.64.f.
  There are also default versions sv_options.32.f and sv_options.64.f that are symbolic links to sv_1.1_options.32.f and sv_1.1_options.64.f correspondingly.
- ***e*** options: e_options.32.f , e_options.64.f
- SystemC options: sc_options.32.f , sc_options.64.f

You can find these files in the directory `$UVM_ML_HOME/ml/run_utils/`.

> The above argument files are needed if the language they address is part of the UVM-ML verification environment being simulated.

> SystemVerilog 1.2 is currently supported as an early adopters version and is not fully validated yet.

In these files, all relevant irun options are listed, along with brief comments explaining why they are needed. For more details, see the table in irun Options Used with UVM-ML .

These argument files are used by the provided examples. They can serve you both for learning and for inclusion in your actual environment. Some users prefer to pull out specific options and organize them within their own "compile and run" schemes, while other users may want to refer directly to these argument files.

irun Options Used with UVM-ML

The following table provides information about the various irun arguments and flags used for running UVM-ML. These flags can be used either at the command line or through irun argument files (using the "–f <arg-file>" option). Examples of such argument files can be found in `$UVM_ML_HOME/ml/run_utils/` . The usage of these argument files is explained in the section on "The Provided irun Argument Files".

Notice that some of the arguments are needed only for specific language combinations, and some arguments are optional. When names or paths depend on tool versions or bit platforms, these are highlighted by colored text.

irun Options	Description
Main Options (for all languages)	
`-ml_uvm`	Indicates to Incisive this is a UVM-ML run.

`+UVM_TESTNAME=[<fw-id>:]<test-name>`	Specify the name of the test to run. The **<fw-id>** is a framework identifier, such as SV (or UVMSV), e, or SC. If no “<fw-id>:” is specified, the “sv:” identifier is implicitly added (to align with the way this option is used by UVM-SV). The **<test-name>** in SV and SC this is a test class, for ***e*** this is test file. This option, as with UVM for SV, can be used for UVM-ML, with the addition of the " <fw-id>:" prefix. For example: `+UVM_TESTNAME=SV:test5` or `+UVM_TESTNAME=e:./test7.e` Use this option **in conjunction** with using uvm_ml_run_test() in the SV code. Similarly to the behavior of the UVM-SV run_test, if both +UVM_TESTNAME and the “test” argument of uvm_ml_run_test() are specified, +UVM_TESTNAME value wins.
`-uvmtest <fw-id>:<test-name>`	Specify the test name. See the explanation of <fw-id>:<test-name> above (there is no implicit <fw-id> value, it must be specified). For example: `-uvmtest SV:test5` or `-uvmtest e:./test7.e` Use this option when uvm_ml_run_test() is **not used** in the SV code.

`-uvmtop <fw-id>:<top-name>`	Declare the top components, for side-by-side parallel trees. See the explanation of <fw-id> above (there is no implicit <fw-id> value, it must be specified) . The <top-name> f or SV or SC is a class name, while for ***e*** this is an ***e*** file that will be loaded (leading to units instantiated under sys). For example: `-uvmtop SC:sctop` or `-uvmtop SV:./svtop.sv` Use this option when uvm_ml_run_test() is **not used** in the SV code.
`-uvmhome ${UVM_ML_HOME}/ml/frameworks/uvm/sv/[1.1d-ml|1.2-ml]`	Point to the location of the UVM-SV ML enabled library, and to the UVM-SC library.
`-f ${UVM_ML_HOME}/ml/run_utils/ml_options.[32|64].f`	A general irun ML arg-file, customized for either 32 or 64 bit platforms. It groups some of the options above for ease of use. You can use it in your own environments, like the ML examples do, or explicitly use selected options.
e Options	
`-snshlib ${UVM_ML_HOME}/ml/libs/uvm_e/${IES_VERSION}/${UVM_ML_COMPILER_VERSION}/[64bit/]libsn_sn_uvm_ml.so`	Point to the pre-compiled UVM-e ML adapter library, customized for an IES version (e.g., 14.1, or any supported version), a gcc version (4.1 or 4.4) , and a platform bit-width (the default is a lib for 32 bit; for 64 bit add the "64bit/" directory).
`-f ${UVM_ML_HOME}/ml/run_utils/e_options.[32|64].f`	An ***e*** related irun arg-file, customized for either 32 or 64 bit platforms, grouping some ***e*** options. You can use it in your own tests, like the examples do, or explicitly use selected options.
SystemVerilog Options	

`-sv_lib $UVM_ML_HOME/ml/libs/uvm_sv/ ${UVM_ML_COMPILER_VERSION}/ [64bit/]libuvm_sv_ml.so`	libuvm_sv_ml.so is a shared library customized for the selected gcc version (4.1 or 4.4) and platform bit-width (the default is 32 bit, add "64bit/" dir for 64 bit). This library contains the DPI C code for the UVM-SV ML adapter.
`-uvmexthome ${UVM_ML_CDS_INST_DIR}/tools/methodology/UVM/CDNS-[1.1d|1.2]`	Points to Cadence UVM SV extensions. By default, if not exist, the 1.1d related extensions are used. If UVM SV 1.2 is used, -uvmexthome must be explicitly specified. The environment variable UVM_ML_CDS_INST_DIR is automatically set by the UVM-ML OA installer and its derivative setup script.
`-f ${UVM_ML_HOME}/ml/run_utils/ sv_[1.1|1.2]_options.[32|64] .f`	SV related irun arg-file. It groups some of the SV options above, customized for the selected UVM version and bit-width. You can use it as is, or use the options in it explicitly. Note that by selecting a specific f file, you actually select the UVM-SV version you want to use. There are also default sv_options.[32|64].f files that point to the 1.1 files.
SystemC Options	
`-sysc`	This option should come first before all other SystemC related options (to enable SC usage).
`-gcc_vers ${UVM_ML_COMPILER_VERSION}`	Select the gcc version (4.1 or 4.4); default version is 4.4. (optional).
`-DSC_INCLUDE_DYNAMIC_PROCESSES`	Required for proper operation of UVM-SC.
`-I${UVM_ML_HOME}/ml/ frameworks/uvm/sc`	SC include directory (for the UVM-SC framework).

`-I${UVM_ML_HOME}/ml/adapters/uvm_sc`	SC include directory (for the UVM-SC ml adapter).
`-I${UVM_ML_HOME}/ml/adapters/uvm_sc/common`	SC include directory (for the UVM-SC ml adapter).
`-I${UVM_ML_HOME}/ml/adapters/uvm_sc/ncsc`	SC include directory(for the UVM-SC ml adapter).
`-spec ${UVM_ML_HOME}/ml/tools/specfiles/${IES_VERSION}/specfile.lnx86.gnu.${UVM_ML_COMPILER_VERSION}[.64bit]`	Point to a 'specfile', customized for an IES version , a gcc version , and a platform bit-width (the default is 32 bit; for 64 bit add the ".64bit" suffix). The specfile lists various options to be passed by irun/ncsc_run to the compiler. (optional)
`-L${UVM_ML_HOME}/ml/libs/ncsc/${IES_VERSION}/${UVM_ML_COMPILER_VERSION}[/64bit/]`	Point to a directory to search for libraries (libraries stated with "-l<short name>" will be looked for in these "-L<dirname>"). This directory is customized for an IES version, a gcc version, and a platform bit-width (the default is 32 bit; for 64 bit add the "/64bit/" directory). (optional)
`-f ${UVM_ML_HOME}/ml/run_utils/sc_options.[32|64].f`	SC related irun arg-file, customized for either 32 or 64 bit platforms. It groups some of the SC options above; you can use it in your own tests, like the examples do, or explicitly use selected options.
Other Options (general, optional)	
`-define USE_UVM_ML_RUN_TEST`	Used in the ML examples to enable procedural invocation (users need not mimic this) (optional)

More details about each of these options can be found in the Cadence Help for irun (and in other Cadence Help books). Here we provide only a listing of the options specifically used in context of UVM-ML, with a brief description.

SystemVerilog 1.2 is currently supported as an early adopters version and is not fully validated yet.

Running Incisive Using irun (typical usage)

To run a simulation with three frameworks, UVM-SV, UVM-e, and UVM-SC, on a 32 bit platform using UVM-SV1.1d, you would typically issue an irun command such as the following:

```
% irun ./svtop.sv ./env.e ./scenv.cpp -uvmtest SV:test17 \
-f $UVM_ML_HOME/ml/run_utils/ml_options.32.f \
-f $UVM_ML_HOME/ml/run-utils/sv_1.1_options.32.f \
-f $UVM_ML_HOME/ml/run_utils/e_options.32.f \
-f $UVM_ML_HOME/ml/run_utils/sc_options.32.f
```

The first line lists the source files to compile and selects a test, and the other lines list the –f argument files, one generic for ML and three specific for the languages involved in this run.

If you are compiling for a 64-bit platform, replace `.32.f` with `.64.f` in all the above option files.

To see more variations of compile flows for different language combinations, see the examples under `$UVM_ML_HOME/ml/examples/use_cases/` , and for any of them run the command

```
% demo.sh IES -dry
```

See "Running an Example with 'demo.sh --dry-run'" for more details.

Running Incisive in Multi-Step Compile and Run Flow

The multi-step flow has three main steps: Compile, Elaborate, and Run. The compile step may be split into several parts, for the various languages, therefore there are often more than three steps. The multiple steps flow is also know as 3-step compilation.

There are two ways to invoke Incisive for a multi-step usage flows:

- The first involves using **irun –compile** , **irun –elaborate** and **irun –R** (run).
- The second involves using **ncvlog** to compile, **ncelab** to elaborate, and **ncsim** to run.

This section explains the second flavor, using the basic **ncvlog** , **ncelab** , and **ncsim** commands.

The example used here shows a use case of code in SV and e, where all the e code is compiled. There are some variations on this flow, such as the e code being loaded rather than compiled, or having also SystemC code (in addition or in place of SystemVerilog or ***e***). The variations are explained in the section titled "variations" below.

To run a UVM-ML environment, user code and pre-compiled libraries need to be brought in. You can see the following code and libraries in the example below:

- **User code:** user ***e*** code is imported by t he topmost ***e*** module ***top.e*** (which imports a few other ***e*** modules) and which is compiled in step 0. SystemVerilog code is imported and included by the file ***test.sv*** which is compiled in step 2.
- **The UVM-ML backplane** : the backplane shared library ***lib_uvm_ml_bp.so*** is linked in step 4 automatically, (this shared library is compiled and created during the installation of UVM-ML).
- **UVM-ML adapters** : each framework adapter has a shared library created during installation, which needs to be linked in for the framework to participate. The adapters are linked in step 4 (you can see explicitly the SystemVerilog adapter ***libuvm_sv_ml.so***). However the preparations and some implicit linking is done earlier:
    - The Specman UVM-e adapter, in this example, is linked with the compiled user code in step 1, where ***libsn_my_e_top.so*** i s created and pointed to by the SPECMAN_DLIB env-variable.
    - The UVM-SV adapter comprises of two parts: libuvm_sv_ml.so and the SystemVerilog package uvm_ml. In the single-step flow, irun automatically compiles the uvm_ml package for you. In the multi-step flow, you need to manually compile the package, located at $UVM_ML_HOME/ml/adapters/uvm_sv/uvm_ml_adapter.sv, along with the user's SystemVerilog code, as seen in step 2.
- **Framework libraries** : for running the SystmeVerilog framework, the UVM-SV package ***uvm_pkg.sv*** (enhanced with some ML enablers) is compiled in step 2. The PLI/DPI shared libraries ***libuvmpli.so*** & ***libuvmdpi.so*** are use d in step 4. There is no need for a special ***e*** related library, nor to import explicitly the UVM-e package uvm_e, as all the ***e*** capabilities are already included within Specman.
- Note: When SystemC is involved, you will also use, and see on the invocation lines, the UVM-SC framework and adapter, and if you use the ASI SystemC simulator, you may also see references to the ASI SystemC library.

One last thing to remember is to make sure your current session has the UVM-ML environment set up properly. For that you typically would run one of the setup scripts located, after the package was installed, under the UVM-ML installation directory `$UVM_ML_HOME/ml/setup*`

The steps of the multi-step flow:

The steps below show a normal multi-step compile flow, with some specific additions as mentioned above to pull in the required backplane, framework, and adapters. In the commands shown, the user specific names are shown in bold.

Step #0: Compile the e verification environment

```
% sn_compile.sh ./top.e -o my_e_top -shlib -exe -32 -t tmp \
             -s  ${UVM_ML_HOME}/ml/libs/uvm_e/14.1/4.4/sn_uvm_ml
```

Description : Compile the e files of the given verification environment. This step can be skipped when there are no e files, or if the e files are loaded in the final run step.

Inputs : the top ***e*** file **top.e** and the name chosen for the outputs following the -o **my_e_top**

Flags :

Flag	Description
-o	Output (target) base name: define the name of the .exe and the base name for the .so shared object.
-exe	Generate an executable file, named according to the –o name => **my_e_top** . This executable is needed for the stub file creation. It is not used during simulation.
-shlib	Create a shared library, named according to the -o name: libsn_ + target name => **libsn_my_e_top.so** .
-32|64	Compile for 32 or 64 bit platform. The example above is for 32 bit. Notice that the path in the -s argument below also changes between 32 and 64 bit platforms.

-t	The directory for placing temporary result files.
-s	Specify the Specman executable used for compiling the provided ***e*** files (-s stands for Specman). For UVM-ML, the executable should have the UVM-e ML adapter compiled in it, as is the case for `sn_uvm_ml` used here. The path to `sn_uvm_ml` is slightly different for 32 or 64 bit platforms (and also depends on the IES and GCC ve rsions): • For 32 bits, the path is: `${UVM_ML_HOME}/ml/libs/uvm_e /${IES_VERSION}/${UVM_ML_COMPILER_VERSION}/s n_uvm_ml` • F or **64** bit s, the path is: `${UVM_ML_HOME}/ml /libs/uvm_e/${IES_VERSION} / ${UVM_ML_COMPILER_VERSION}/ 64bit/ sn_uvm_ml` (In the example above, we used IES version 14.1 and GCC version 4.4.)

Outputs :

- Specman executable: **my_e_top.exe** – needed for step 1
- Shared Library: **libsn_my_e_top.so** – needed for step 4

Explanation :

This command prepares two things: a shared library object (.so file), and a Specman executable file. We now need to set the environment variable to point to the shared object, as preparation for step 3. If done in a bash shell script, this would be done as follows:

```
SPECMAN_DLI B= ./ libsn_my_e_top.so ; export SPECMAN_DLIB
```

If all the subsequent steps in this section are executed in the same current directory (./), then SPECMAN_PATH need not be defined. Otherwise, the SPECMAN_PATH should also be set. It should point to the directory where compilation (step 0) was done so that the compilation products are found by the following steps.

Compilation of the ***e*** code is needed only if you have a significant amount ***e*** code, and want to maximize performance. A slightly shorter flow (without step 0) is possible if the ***e*** code is loaded. See the variations section below for a more detailed discussion of these two options.

Step #1: Write the Specman Stub file

The stub file is generated using the executable created in step #0

```
% ./my_e_top -c 'write stub -ncsv my_ml_nc_stubs.sv '
```

Description : Write a Specman stub file, for communication with the simulator and for resolving HDL paths.

Inputs : the name given to the stub file: **my_ml_nc_stubs.sv** (a file bearing this name will be the output)

Flags :

Flag	Description
-c	Provide a Specman command. In this case, the command passed is `'write stub -ncsv <stub-name>'` for writing a stub file to be used by Incisive

Output : the Specman SV stub file **my_ml_nc_stubs.sv**

Explanation :

The Specman executable **./my_e_top** was created in step 0. This command, with the option -ncsv, creates a stub file aimed for use with with Incisive and a SystemVerilog DUT.
If the DUT is mixed-language (Verilog + VHDL) you may need to create multiple stubs;

Step #2: Compile SystemVerilog files

```
% ncvlog -sv -messages \
     $UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/src/uvm_pkg.sv \
      $UVM_ML_HOME/ml/adapters/uvm_sv/uvm_ml_adapter.sv \
      -incdir $UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/src \
      -incdir $UVM_ML_HOME/ml/adapters/uvm_sv \
      ./test.sv ./my_ml_nc_stubs.sv
```

Description : Compile all the SystemVerilog files, using the Incisive ncvlog command

Inputs : the files compiled come from two categories:

- User files: the SystemVerilog files **test.sv** (user code) and the stub **my_ml_nc_stubs.sv** (generated in step 1)
- UVM packages: the ML enabled UVM-SV framework **uvm_pkg.sv** , and its ML adapter **uvm_ml_adapter.sv** (these packages reside in the UVM-ML installation) .
- Note that if you want to use SystemVerilog 1.2, you need to change "1.1d" in the path of uvm_pkg.sv path to "1.2".

Flags :

Flag	Description
-sv	Indicates there are SystemVerilog files (not only Verilog).
-messages	Print out informative messages.
-incdir	Specify directories in which to search for " include" files.

Outputs : The complied SystemVerilog modules (located in INCA_libs) are listed towards end of the irun printout:

```
file: ./test.sv
          module worklib.topmodule
file: ./ml_nc_stubs.sv
          module worklib.specman
          module worklib.specman_wave
```

Explanation :The two SystemVerilog packages **uvm_pkg.sv** and **uvm_ml_adapter.sv** must be compiled first. We first compile an ML-enabled version of the UVM-SV library, which resides within the UVM-ML installation. Then we compile the adapter that enables communication of the UVM-SV framework with the ML backplane. After these, we compile the user files.

> SystemVerilog 1.2 is currently supported as an early adopters version and is not fully validated yet.

Step #3: Elaboration

```
% ncelab -messages worklib.topmodule worklib.specman worklib.specman_wave
```

Description : Elaborate all the HDL modules, using the Incisive **ncelab** command

Inputs :All the modules compiled by ncvlog in step 2: **worklib.topmodule** **worklib.specman** **worklib.specman_wave**

Outputs : Various outputs are placed into the ./INCA_lib directory

Step #4: Run the simulation

```
% ncsim worklib.topmodule  -uvmtop  SV:svtop  \
      -sv_lib `ncroot`/tools/uvm/uvm_lib/uvm_sv/lib/libuvmpli.so \
      -sv_lib `ncroot`/tools/uvm/uvm_lib/uvm_sv/lib/libuvmdpi.so \
      -sv_lib ${UVM_ML_HOME}/ml/libs/uvm_sv/4.4/libuvm_sv_ml.so \
      -run -exit -messages
```

Description : Run the simulation using the Incisive **ncsim** command

Inputs :The elaboration snapshot, which contains **worklib.topmodule** and the user top module **SV:svtop**

Flags :

Flag	Description
-sv_libs	SystemVerilog shared libraries that need to be pulled in: - **libuvmpli.so** and **libuvmdpi.so** – needed for the UVM cadence additions (for UVM-SV). Are provided as part of the IES release package. - **libuvm_sv_ml.so** – the C/DPI part of the SV ML adapter, created when installing UVM-ML.
-run	Execute the simulation after initialization without waiting for user input.
-exit	Exit the simulation when done running (rather than entering interactive mode).
-messages	Print out informative messages.

Outputs : The outcome is the actual run, with resulting log files, etc.

Explanation : This step brings together all the products of previous steps.
For the e products to be found, SPECMAN_DLIB environment variable needs to be set (it was set in step 0). This enables NCSIM to load the shared object which contains the ***e*** ML adapter and the compiled ***e*** code.
SPECMAN_PATH may also need to be set for this step, if the Specman products are not in the current working directory. In this example, since they are located here, it needs not be set .
Also, in this step ***lib_uvm_ml_bp.so*** is automatically loaded if the command line contains any of the flags "-uvmtop", "-uvmtest" or "-ml_uvm" (**at least** one of them should used).

Multi-Step Variations

There are many possible variations to the 4-5 steps listed above, depending on your needs, setup, or simulator used:

(I) lib_uvm_ml_bp.so
In step 4 above, ***lib_uvm_ml_bp.so*** is automatically and **implicitly** pulled in. This is done automatically for the IES simulator. Other simulators would need to **explicitly** pull it in. You can see the list of all library files in the section on UVM-ML Libraries .

(II) The flow is slightly different if *e* files are to be compiled or not

- When the e code is significant in size or makes significant computations, it is worthwhile to compile it. This flow is described above, and can also be found in the file ies_multistep_non_irun_e_comp.sh which demonstrates the compile flow (in directory $UVM_ML_HOME/ml/examples/use_cases/side_by_side/sv_e).
- If there is very little ***e*** code, or if there is a file that is frequently edited, it is more convenient to load the ***e*** file. The file ***ies_multistep_non_irun_e_load.sh*** demonstrates that. The main difference in this flow is that step 0 (compilation) is omitted, and thus the names of the Specman related files (executable and shared libraries) are different, you can compare the two above files for details.
- A very typical situation is when a majority of the ***e*** code should be compiled, and yet there is a final ***e*** test file that we want to load on top. For that case, use the compilation flow, and just add in the final step the option ***-uvmtest e:my_test_file.e*** to the ncsim command (or, if the test itself is not in e but in SV, you could add ***my_top_file.e*** to the invocation line, to be loaded)

(III) The location of the Specman shared library
Notice that for running Incisive with UVM-ML with ***e*** code, you need to inform Incisive where to find the Specman shared library and related files. This is done by setting two environment variables. If the user ***e*** code is loaded, the shared library and files from the UVM-ML installation are used:

```
# extracted from bash file: multi-step-load.sh
SPECMAN_DLIB=${UVM_ML_HOME}/ml/libs/uvm_e/14.1/4.4/libsn_sn_uvm_ml.so;
export SPECMAN_DLIB
SPECMAN_PATH=${UVM_ML_HOME}/ml/libs/uvm_e/14.1/4.4;
export SPECMAN_PATH
```

If the user code is compiled, the SPECMAN_DLIB should point to the newly compiled shared lib. For example, in the case of the multi-step-load.sh example, this is how it is done:

```
# extracted from bash file: multi-step-comp.sh
SPECMAN_DLIB=./libsn_my_e_top.so; export SPECMAN_DLIB
```

Also, the above examples fit compilation for a 32 bit platform, in which .so libraries are located in

```
${UVM_ML_HOME}/ml/libs/uvm_e/14.1/4.4/
```

If the compilation is for a 64 bit platform, the location of the.so libraries is slightly different

```
${UVM_ML_HOME}/ml/libs/uvm_e/14.1/4.4/64bit/
```

> In the compile flow, if the directory in which the compilation products are created is the current work directory (./), then SPECMAN_PATH need not be defined. Otherwise, the SPECMAN_PATH should point to the directory where compilation was done.

(IV) Other flows
Other multi-step flows, including some that use SystemC, can be seen by running "demo.sh IES -steps" in the various side-by-side examples. Run "demo.sh -help" for more details.

Running Incisive with ASI SystemC

The previous sections explained how to use Incisive as the simulation engine for all three languages: SystemVerilog, SystemC, and ***e*** . That is the typical and simple way to use Incisive. It is also possible to run Incisive as the simulation engine for SystemVerilog and ***e*** , combined with the ASI-SystemC engine for SystemC. This section describes this less common mode.

> ASI SystemC was formerly known as OSCI SystemC. ASI stands for the "Accellera Systems Initiative", a not-for profit organization dedicated to create, support, promote, and advance system-level design, modeling, and verification standards.

When using ASI SystemC, users should first set the following two environment variables:

1. OSCI_SRC – the directory where the ASI SystemC sources are unpacked and located.
   For example:
   ```
   % setenv OSCI_SRC /home/cad/sysc/osci-2.3.0
   ```
   This variable is needed only during installation, for compiling the ASI SystemC code.
2. OSCI_INSTALL - directory where the ASI SystemC simulator has been installed.
   For example:
   ```
   % setenv OSCI_INSTALL /home/cad/osci-gcc.4.4.5
   ```
   There should be in this location two directories, one called lib-linux64 (for 64 bit platforms), and one called lib-linux (for 32 bit platforms).
   This variable is required for every run of with ASI SystemC.

To run with UVM-ML, ASI SystemC needs a small enabler patch. This patch is part of the UVM-ML

release, it is compiled on top of the ASI SystemC installation during the UVM-ML installation and is stored locally in the UVM-ML directories (the original ASI SystemC installation is not modified).

Running an ASI SystemC example with demo.sh:

To see the specific flags used for running any of the SystemC examples under `$UVM_ML_HOME/ml/examples/` with ASI SystemC, you may go to that directory and run the demo.sh –dry command. For example, you can see the commands used for the example sc_sv_e, by doing:

```
% cd $UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv_e
% demo.sh IES -dry -steps ies_osci_cl_multistep
```

Notice we used here a multi-step compilation flow, by utilizing the –steps option. The argument following the –steps option is the name of predefined make-target, customized for using IES with ASI SystemC in a multi-step flow. See the section "Running Incisive with 'demo.sh --dry-run'" for more details on demo.sh and the –dry or –steps options.

Running an ASI SystemC example with explicit commands:

The following example shows a manual invocation of Incisive with ASI SystemC, involving a SystemVerilog + SytstemC testbench, for the example in the directory $UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv/To use the ASI SystemC simulator combined with Incisive, on a 32 bit platform, do the following:

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

```
g++ -g -fPIC - o liball_osci_ies.so ./sc.cpp -m32 \
    -Xlinker -rpath -Xlinker $UVM_ML_HOME/ml/libs/osci/2.3/4.4/ \
    -Xlinker -Bsymbolic -L$UVM_ML_HOME/ml/libs/osci/2.3/4.4/ \
    -shared -I${OSCI_INSTALL}/include \
    -I${UVM_ML_HOME}/ml/adapters/uvm_sc/ \
    -I${UVM_ML_HOME}/ml/adapters/uvm_sc/osci \
    -I${UVM_ML_HOME}/ml/frameworks/uvm/sc -I. \
    -luvm_sc_fw_osci -luvm_sc_ml_osci
```

> The directory paths `/2.3/4.4/` represent /<OSCI version>/<g++ version>. You can use any of the supported ASI SystemC library versions (for example, 2.2, 2.3, as indicated by the libs/osci/* sub-directories available in the installation). The <g++ version> should be one of the supported compiler versions (for example, 4.1 or 4.4), and the same version used when installing UVM-ML.

Inputs and flags :

Many of the lines above are generic for ASI SystemC compilation. However, the options in blue

text relate specifically to running UVM-ML, and have the following significance:

Flag	Description
`-o liball_osci_ies.so`	The output of this compilation will be the shared object liball_osci_ies.so.
`./sc.cpp`	The user C++ code file(s), in this simple example, just one file.
`-I$UVM…/adapters/…`	Specify include directories, in this case where the UVM-ML SC adapter code is.
`-I$UVM…/frameworks/…`	Specify include directories, in this case where the UVM-ML SC framework code is.
`-I$UVM…/libs/osci/…`	The location of the relevant shared objects and link files (the -Xlinker and -L options).
`-luvm_sc_fw_osci`	The shared object of the uvm-sc framework (in the -L location).
`-luvm_sc_ml_osci`	The shared object of osci (ASI SystemC) lib and the uvm-sc adapter (in -L loc.)

Output : The output is the shared object to be used in the next ste p: liball_osci_ies.so

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

```
irun ./test.sv -incdir . -top topmodule \
     -f $UVM_ML_HOME/ml/run_utils/ml_options.32.f \
     -f $UVM_ML_HOME/ml/run_utils/sv_1.1_options.32.f \
     -define USE_UVM_ML_RUN_TEST \
     -L`pwd`liball_osci_ies.so \
     -access +rw
```

Inputs and flags :

The lines in blue relate specifically to UVM-ML, and have the following significance:

Flag	Description
`./test.sv`	User code file(s), in this simple example, just one file.

`-f $UVM_ML_HOME/…options.f`	Specify argument files, one generic and one for SV.
`-define USE_UVM_ML_RUN_TEST`	Needed for the test to start running in procedural mode.
`liball_osci_ies.so`	The shared object created in step 1.
-access +rw	Turn on read, write access to HDL signals. This flag is currently necessary to enable automated synchronization between the simulator and ASI SystemC framework.

Using the Makefile

Alternatively, you can invoke any of the SC examples under `$UVM_ML_HOME/ml/examples/` with ASI SystemC by using the makefiles included in the release. If you invoke those examples, there are different Make targets there that you can use to compile IES with ASI SystemC in various modes:

`% make ies_osci_proc` - IES with ASI SystemC, procedural invocation mode

`% make ies_osci_cl_multistep` - IES with ASI SystemC, command line invocation, multi-step flow

The full list of IES ASI SystemC targets can be found in the section The Underlying Infrastructure: UVM-ML Makefiles (or in the Makefiles themselves).

Running UVM-ML with VCS

This section shows how to compile and run a UVM-ML environment with the Synopsys VCS Simulator. In the examples shown here, VCS is the SystemVerilog simulation engine. It is combined with Specman when ***e*** is also involved, and with ASI SystemC when SystemC is involved.

Section Contents

Running an Example with demo.sh

The easiest way to see an example of the compile and run commands is by using the demo.sh script with the `--dry-run` option, as explained in Using 'demo.sh --dry-run' .

The output from running `demo.sh VCS -dry` is shown below. This output was shortened for clarity (the '...' markers show where text was removed). You should run the command interactively to get the full printout.

> The information printed to the screen can be placed into a file and then run "as is", as a shell script. You can also edit this file and modify it to your needs.

> The examples make use of specific version:
>
> - 14.1 is the Specman version. It can be replaced with `$IES_VERSION` or any supported IES version.
> - 4.4 is the gcc version. It can be replaced with `$UVM_ML_COMPILER_VERSION` or any supported gcc version.
> - 2.3 is the ASI SystemC version (previously known as OSCI SystemC). It can be replaced by any supported ASI SystemC version.

In this section we will use the `/side_by_side/sc_sv_e` example for demonstration.

The `demo.sh` script makes use of the environment variables `UVM_ML_CC` and `UVM_ML_LD`:

- These environment variables are defined in the setup scripts created during the installation of the UVM-ML package and located in `$UVM_ML_HOME/ml/setup*` . They should point to the location of the preferred g++ installation. Typically this is within the installation directories of the chosen simulator.
- `UVM_ML_CC` should contain a reference to the user-preferred C++ compiler and optionally the accompanying parameters, to be used in C++ source file compilations. It is semantically similar to 'CC' / 'CXX' variables often used in makefiles.
- `UVM_ML_LD` should contain a reference to the user-preferred C++ compiler and optionally the accompanying parameters, to be used while linking together the UVM-ML-OA shared libraries and executables.

Running VCS with Specman

This section uses the example found in `$UVM_ML_HOME/ml/examples/use_cases/side_by_side/sv_e/` .

To use the VCS simulator and Specman in interpreted mode, on a 64 bit platform, do the following:

Preparation : the following environment variables should be set

```
% setenv VCS_UVM_HOME $UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/src
% setenv SPECMAN_DLIB $UVM_ML_HOME/ml/libs/uvm_e/14.1/4.4/libsn_sn_uvm_ml.so
% setenv SPECMAN_PATH $UVM_ML_HOME/ml/libs/uvm_e/14.1/4.4:.:
```

1. Create a Specman SV stub file for VCS:

```
% specrun -dlib -64 -c "load top.e; write stub s -vcssv"
```

Output : The output is the Specman stub file specman.sv (the file name is the default name)

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():

```
% vcs -o simv64 \
      -CFLAGS ' -O0 -g' -LDFLAGS ' -O0 -g -Wl,-E' \
      -sverilog +acc +vpi -timescale=1ns/1ns -ntb_opts uvm-1.1 -full64 -debug \
      +incdir$VCS_UVM_HOME/seq +incdir$VCS_UVM_HOME +incdir$VCS_UVM_HOME/tlm2 \
      +incdir$VCS_UVM_HOME/base +incdir. \
```

```
+define+CDNS_EXCLUDE_UVM_EXTENSIONS +define+CDNS_RECORDING_SVH \
+incdir$UVM_ML_HOME/ml/adapters/uvm_sv \
+define+USE_UVM_ML_RUN_TEST \
$UVM_ML_HOME/ml/adapters/uvm_sv/uvm_ml_adapter.sv \
test.sv specman.sv \
$UVM_ML_HOME/ml/libs/backplane/4.4/64bit/libuvm_ml_bp.so \
$UVM_ML_HOME/ml/libs/uvm_sv/4.4/64bit/libuvm_sv_ml.so \
 -P `sn_root -home`/src/pli.tab `sn_root -dir -64`/libvcssv_sn_boot.so
```

Inputs and flags : Most of the lines above are generic use of VCS. However, the lines in blue text relate specifically to running UVM-ML, and have the following significance:

Flag	Description
`+incdir$UVM_ML…/adapters/uvm_sv`	indicates the UVM-ML adapter for SV is located here
`+define+USE_UVM_ML_RUN_TEST`	is needed for the test to start running
`$UVM…/uvm_sv/uvm_ml_adapter.sv`	the code of the UVM-ML adapter for SV
`test.sv`	user code file(s), in this simple example, just one file
`specman.sv`	and the Specman stub generated in step 1 above
`$UVM_ML…/libuvm_ml_bp.so`	The backplane shared library
`$UVM_ML…/libuvm_sv_ml.so`	The DPI-C code of the SV adapter (shared library).

> The directory pat hs /4.4/ re present the <g++ version> version used. You can use any of the supported compiler versions (for example, 4.1 or 4.4), and this should be the same version used when installing UVM-ML.

Output : The output of this command is the executable object to be run in the next step: simv64

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

```
% echo "run" > vcs_input.do
% echo "exit" >> vcs_input.do
% ./simv64 -ucli -i vcs_input.do
```

The first two lines create an indirect file with the commands for VCS to run and then exit. The final lines invokes the executable created in step to, with this file.

Running VCS with ASI SystemC

> ASI SystemC was formerly known as OSCI SystemC. ASI stands for the "Accellera Systems Initiative", a not-for profit organization dedicated to create, support, promote, and advance system-level design, modeling, and verification standards.

When using ASI SystemC, users should first set the following two environment variables:

1. OSCI_SRC – the directory where the ASI SystemC sources are unpacked and located.
   For example:
   % setenv OSCI_SRC /home/cad/sysc/osci-2.3.0
   This variable is needed only during installation, for compiling the ASI SystemC code.
2. OSCI_INSTALL - directory where the ASI SystemC simulator has been installed.
   For example:
   % setenv OSCI_INSTALL /home/cad/osci-gcc.4.4.5
   There should be in this location two directories, one called lib-linux64 (for 64 bit platforms), and one called lib-linux (for 32 bit platforms).
   This variable is required for every run of with ASI SystemC.

To run with UVM-ML, ASI SystemC needs a small enabler patch. This patch is part of the UVM-ML release, it is compiled on top of the ASI SystemC installation during the UVM-ML installation and is stored locally in the UVM-ML directories (the original ASI SystemC installation is not modified).

Running an ASI SystemC example with explicit commands:

To run with UVM-ML, ASI SystemC needs a small enabler patch. This patch is part of the UVM-ML release, it is compiled on top of the ASI SystemC installation during the UVM-ML installation and is stored locally in the UVM-ML directories (the original ASI SystemC installation is not modified)

This section uses the example found in
`$UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv/.`

To use VCS with the ASI SystemC simulator, on a 64 bit platform, do the following:

1. First, compile the SystemC code and create a combined library with the ASI SystemC simulator, and with the UVM-SC framework and adapter libraries (that is, the ASI SystemC flavors of the UVM-SC framework and the UVM-ML adapter for UVM-SC):

```
% g++ -g -fPIC -o liball_osci_vcs.64.so \
    -I$OSCI_INSTALL/include \
    -I$UVM_ML_HOME/ml/adapters/uvm_sc \
    -I$UVM_ML_HOME/ml/adapters/uvm_sc/osci \
    -I$UVM_ML_HOME/ml/frameworks/uvm/sc \
    –shared -Xlinker -rpath \
```

```
-Xlinker $UVM_ML_HOME/ml/libs/osci/2.3/4.4/64bit \
-L$UVM_ML_HOME/ml/libs/osci/2.3/4.4/64bit \
-luvm_sc_fw_osci -luvm_sc_ml_osci sc.cpp
```

> The directory paths /2.3/4.4/ represent /<OSCI version>/<g++ version>. You can use any of the supported ASI SystemC library versions (for example, 2.2, 2.3, as indicated by the libs/osci/* sub-directories available in the installation). The <g++ version> should be one of the supported compiler versions (for example, 4.1 or 4.4), and the same version used when installing UVM-ML.

Inputs and flags : Most of the lines above are generic for ASI SystemC compilation. However, the lines in blue text relate specifically to running UVM-ML, and have the following significance:

Flag	Description
`-I$UVM_ML…/ml/adapters/…`	include directories, where the UVM-ML SC adapter code is
`-I$UVM…/ml /frameworks/…`	include directories, where the UVM-ML SC framework code is
$UVM_ML…/ml/libs/osci/…	The location of the relevant shared objects and link files (stated in two lines, see both the –Xliner and the –L options)
`-luvm_sc_fw_osci`	The shared object of the uvm-sc framework (in the –L location)
`-luvm_sc_ml_osci`	The shared object of ASI SystemC lib and the uvm-sc adapter (in –L loc.)
`sc.cpp`	The user C++ code file(s), in this simple example, just one file

Output : The output is the shared object to be used in the next step: liball_osci_vcs.64.so

2. Next, VCS is invoked to compile the SystemVerilog files and link them with the UVM-ML backplane, and with the library created in step 1 above:

```
% setenv VCS_UVM_HOME $UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/src

% vc s -o simv64 \
      -CFLAGS ' -O0 -g' -LDFLAGS ' -O0 -g -Wl,-E' -full64 \
      -sverilog +acc +vpi -timescale=1ns/1ns -ntb_opts uvm-1.1 \
      +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 \
+define+USE_UVM_ML_RUN_TEST \
$UVM_ML_HOME/ml/adapters/uvm_sv/uvm_ml_adapter.sv test.sv \
$UVM_ML_HOME/ml/libs/backplane/4.4/64bit/libuvm_ml_bp.so \
$UVM_ML_HOME/ml/libs/uvm_sv/4.4/64bit/libuvm_sv_ml.so \
liball_osci_vcs.64.so
```

The directory paths /4.4/ re present the <g++ version> version used. You can use any of the supported compiler versions (for example, 4.1 or 4.4), and this should be the same version used when installing UVM-ML.

Inputs and flags : The lines in blue relate specifically to UVM-ML, and have the following significance:

Flag	Description
`+incdir$UVM_ML…/adapters/uvm_sv`	indicates the UVM-ML adapter for SV is located here
`+define+USE_UVM_ML_RUN_TEST`	is needed for the test to start running in proc. mode
`$UVM…/uvm_sv/uvm_ml_adapter.sv`	the code of the UVM-ML adapter for SV
`test.sv`	user code file(s), in this simple example just one file
`$UVM_ML…/libuvm_ml_bp.so`	The backplane shared library
`$UVM_ML…/libuvm_sv_ml.so`	The DPI-C code of the SV adapter (shared library)
liball_osci_vcs.64.so	The shared object created in step 1 (ASI SystemC + SC ML libs)

Output : The output of this command is the executable object to be run in the next step: simv64

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

```
% ./simv64
```

Running UVM-ML with Questa

This section shows how to compile and run a UVM-ML environment with the Mentor Questa Simulator. In the examples shown here, Questa is the SystemVerilog and SystemC simulation engine, Specman is the simulation engine for ***e*** when ***e*** involved, and is some cases ASI SystemC is also combined as an alternative SystemC simulation engine.

There are several ways to run the UVM-ML examples with Questa. We start with the simplest way, using the `demo.sh` script, using the ASISystemC simulation engine for SystemC. Then we go through an alternative way, using the Questa native SystemC engine.

About installation: When you use the provided scripts to install UVM-ML for Questa, make sure to use `install_questa.csh` (or alternatively, `install.csh --no-ncsc …`). For information on installation and the install scripts, see `$UVM_ML_HOME/ml/README_INSTALLATION.txt`

About using Questa with SystemC: To use Questa with SystemC you should set the two environment variables UVM_ML_CC and UVM_ML_LD to point to the g++ location in the Questa installation directory:

- `UVM_ML_CC` should contain a reference to the user-preferred C++ compiler and optionally the accompanying parameters, to be used in C++ source file compilations. It is semantically similar to 'CC' / 'CXX' variables often used in makefiles.
- `UVM_ML_LD` should contain a reference to the user-preferred C++ compiler and optionally the accompanying parameters, to be used while linking together the UVM-ML-OA shared libraries and executables.

These environment variables are defined in the setup scripts created during the installation of the UVM-ML package and located in `$UVM_ML_HOME/ml/setup*` .

About using the native Questa SystemC : SystemC users can run Questa in two modes: with the native SystemC engine of Questa, or linked with the ASI SystemC engine.

For using the native Questa SystemC, the UVM-ML **portable adapter** needs to be linked in. To use the portable adapter with Questa use "make questa_sc", or do it manually according to the steps described in Manual Usage of the Portable Adapter .

The portable adapter has a few limitations: it does not support global timeout, and unified hierarchy works only with SystemC on top of ***e*** or SystemVerilog.

Section Contents

Running an Example with demo.sh

The easiest way to see an example of the compile and run commands is by using the demo.sh script with the `--dry-run` option, as explained in Using ‘demo.sh --dry-run’ .

The output from running `demo.sh QUESTA -dry` is shown below. This output was shortened for clarity (the '...' markers show where text was removed). You should run the command interactively to get the full printout.

> The information printed to the screen can be placed into a file and then run "as is", as a shell script. You can also edit this file and modify it to your needs.

> The examples make use of specific version:
>
> - 14.1 is the IES version. It can be replaced with $IES_VERSION or any supported IES version.
> - 4.4 is the gcc version. It can be replaced with $UVM_ML_COMPILER_VERSION or any supported gcc version.
> - 2.3 is the ASI SystemC version. It can be replaced by any of the supported versions.

In this section we will use the `/side_by_side/sc_sv_e` example for demonstration.

```
      % cd ${UVM_ML_HOME}/ml/examples/use_cases/side_by_side/sc_sv_e/
      % demo.sh QUESTA -dry
      #!/bin/sh
      # Dry-run output: the command(s) used to run this test are:
      ...
      # questa_library_preparation_command :
      vlib work
      ...
      # questa_compile_command :
      vlog -dpicppinstall
      /cad/tools/cadence/IUS14.1_latest/linux/tools/cdsgcc/gcc/4.4/inst
      all/bin/g++ -suppress 2218,2181 +define+USE_UVM_ML_RUN_TEST
      +incdir${UVM_ML_HOME}/ml/frameworks/uvm/sv/1.1d-ml/src ...
      ...
      # specman_compile_command :
      sn_compile.sh -s ${UVM_ML_HOME}/ml/libs/uvm_e/14.1/4.4/sn_uvm_ml top.e -o libsn_top.so -
shlib
      # osci_compile_command :
      ${UVM_ML_LD} -g -fPIC -o liball_osci_questa.so sc.cpp -m32 -Xlinker
      -rpath -Xlinker ${UVM_ML_HOME}/ml/libs/osci/2.3/4.4/
      -L${UVM_ML_HOME}/ml/libs/osci/2.3/4.4/ -luvm_sc_fw_osci
      -luvm_sc_ml_osci -shared -I${OSCI_INSTALL}/include
      -I${UVM_ML_HOME}/ml/adapters/uvm_sc/ ...
      ...
      # questa_run_command :
      vsim -c -sv_lib ./libuvm_dpi
      -gblso ${UVM_ML_HOME}/ml/libs/backplane/4.4/libuvm_ml_bp.so
      -sv_lib ${UVM_ML_HOME}/ml/libs/uvm_sv/4.4/libuvm_sv_ml
      -gblso ${UVM_ML_HOME}/ml/libs/uvm_sv/4.4/libuvm_sv_ml.so ...
      ...
```

Running Questa with Specman

This section uses the example found in `$UVM_ML_HOME/ml/examples/use_cases/side_by_side/sv_e/`.

To use the Questa simulator and Specman in interpreted mode, on a 32-bit platform, do the following:

Environment Variables : the following environment variables should be set

```
setenv UVM_HOME $UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml
setenv SPECMAN_DLIB $UVM_ML_HOME/ml/libs/uvm_e/14.1/4.4/libsn_sn_uvm_ml.so
setenv SPECMAN_PATH $UVM_ML_HOME/ml/libs/uvm_e/14.1/4.4:.:
setenv MTI_HOME <path-to-mti-install> # e.g., /tools/mti/10.3/modeltech
```

Execution steps :

1. DPI compilation

```
% g++ -m32 -g -fPIC -I$MTI_HOME/include -shared \
    - o libuvm_dpi.so $UVM_HOM E/src/dpi/uvm_dpi.cc
```

2. Questa library preparation

```
% vlib ./work
```

3. Create a Specman SV stub file for Questa:

```
% specrun -dlib -c "write stub -mti_sv";
```

Output : The output is the Specman stub file specman.sv (the file nam e is the default name)

4. 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():

```
% vlog -dpicppinstall g++ -sv +acc -timescale 1ns/1ns -suppress 2218,2181 \
       +incdir$UVM_HOME +incdir$UVM_HOME/src +incdir. \
        +incdir$UVM_HOME/tlm2 +incdir$UVM_HOME/base +incdir$UVM_HOME/macros \
        +incdir$UVM_ML_HOME/ml/adapters/uvm_sv \
       +define+USE_UVM_ML_RUN_TEST \
       $UVM_HOME/src/uvm.sv \
       $UVM_ML_HOME/ml/adapters/uvm_sv/uvm_ml_adapter.sv \
       ./test.sv specman.sv
```

> The g++ used should be one of the supported compiler versions (e.g., 4.1 or 4.4), and the same version used during installation of UVM-ML.

Inputs and flags : Most of the lines above are generic for Questa and UVM SystemVerilog. However, the lines in blue text relate specifically to running UVM-ML, and have the following significance:

Flag	Description
`+incdir$UVM_ML…/adapters/uvm_sv`	indicates the UVM-ML adapter for SV is located here
`+define+USE_UVM_ML_RUN_TEST`	is needed for the test to start running

`$UVM…/uvm_sv/uvm_ml_adapter.sv`	the code of the UVM-ML adapter for SV
`./test.sv`	user code file(s), in this simple example, just one file
`specman.sv`	and the Specman stub generated in step 3 above

5. The simulation run is launched in the following way:

A Questa input file is prepared

```
% echo "sn set uvm_ml" > mti_input.do
% echo "run -all" >> mti_input.do
% echo "exit -f" >> mti_input.do
```

And VSIM is launched

```
% vsim -c -sv_lib ./libuvm_dpi \
       -gblso $UVM_ML_HOME/ml/libs/backplane/4.4/libuvm_ml_bp.so \
       -sv_lib $UVM_ML_HOME/ml/libs/uvm_sv/4.4/libuvm_sv_ml \
       -gblso  $UVM_ML_HOME/ml/libs/uvm_sv/4.4/libuvm_sv_ml.so \
       -pli `sn_root`/specman/linux/libmti_sv_sn_boot.so \
        -sv_lib `sn_root`/specman/linux/libmti_sv_sn_boot \
        -dpioutoftheblue 1 topmodule specman < mti_input.do
```

Inputs and flags : Most of the lines above are generic for Questa and UVM SystemVerilog. However, the lines in blue text relate specifically to running UVM-ML, and have the following significance:

Flag	Description
`./libuvm_dpi`	DPI shared objected, prepared in step 1
`$UVM_ML…/libuvm_ml_bp.so`	The backplane shared library (from ML install)
`$UVM_ML…/libuvm_sv_ml.so`	The DPI-C code of the SV adapter (from ML install) (stated in two lines, see both the -sv_lib and the -gblso options)

Running Questa with ASI SystemC

> ASI SystemC was formerly known as OSCI SystemC. ASI stands for the "Accellera Systems Initiative", a not-for profit organization dedicated to create, support, promote, and advance system-level design, modeling, and verification standards.

When using ASI SystemC, users should first set the following two environment variables:

1. OSCI_SRC – the directory where the ASI SystemC sources are unpacked and located.
   For example:
   % setenv OSCI_SRC /home/cad/sysc/osci-2.3.0
   This variable is needed only during installation, for compiling the ASI SystemC code.
2. OSCI_INSTALL - directory where the ASI SystemC simulator has been installed.
   For example:
   % setenv OSCI_INSTALL /home/cad/osci-gcc.4.4.5
   There should be in this location two directories, one called lib-linux64 (for 64 bit platforms), and one called lib-linux (for 32 bit platforms).
   This variable is required for every run of with ASI SystemC.

To run with UVM-ML, ASI SystemC needs a small enabler patch. This patch is part of the UVM-ML release, it is compiled on top of the ASI SystemC installation during the UVM-ML installation and is stored locally in the UVM-ML directories (the original ASI SystemC installation is not modified).

Running an ASI SystemC example with explicit commands:

To run with UVM-ML, ASI SystemC needs a small enabler patch. This patch is part of the UVM-ML release, it is compiled on top of the ASI SystemC installation during the UVM-ML installation and is stored locally in the UVM-ML directories (the original ASI SystemC installation is not modified)

This section uses the example found in
`$UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv/.`

To use Questa with the ASI SystemC simulator, on a 32 bit platform, do the following:

Environment Variables : the following environment variables should be set

```
% setenv UVM_HOME $UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml
% setenv MTI_HOME <path-to-mti-install>   # e.g., /tools/mti/10.3/modeltech
```

(as well as the two ASI SystemC environment variables mentioned earlier, OSCI _SRC and OSCI_INSTALL)

Execution steps :

1. DPI compilation

```
% g++ -m32 -g -fPIC -I$MTI_HOME/include -shared \
      -o libuvm_dpi.so $ UVM_HOME/src/dpi/uvm_dpi.cc
```

2. Questa library preparation

```
% vlib ./work
```

3. Compile the SystemVerilog files:

```
% vlog -dpicppinstall g++ -sv +acc -timescale 1ns/1ns -suppress 2218,2181 \
       +incdir$UVM_HOME +incdir$UVM_HOME/src +incdir. \
        +incdir$UVM_HOME/tlm2 +incdir$UVM_HOME/base +incdir$UVM_HOME/macros \
        +incdir$UVM_ML_HOME/ml/adapters/uvm_sv \
       +define+USE_UVM_ML_RUN_TEST \
       $UVM_HOME/src/uvm.sv \
       $UVM_ML_HOME/ml/adapters/uvm_sv/uvm_ml_adapter.sv \
       ./test.sv
```

> The g++ used should be one of the supported compiler versions (e.g., 4.1 or 4.4), and the same version used during installation of UVM-ML.

Inputs and flags : Most of the lines above are generic for Questa and UVM SystemVerilog. However, the lines in blue text relate specifically to running UVM-ML, and have the following significance:

Flag	Description
`+incdir$UVM_ML…/adapters/uvm_sv`	indicates the UVM-ML adapter for SV is located here
`+define+USE_UVM_ML_RUN_TEST`	is needed for the test to start running
`$UVM…/uvm_sv/uvm_ml_adapter.sv`	the code of the UVM-ML adapter for SV
`./test.sv`	user code file(s), in this simple example, just one file

4. Then, compile the SystemC code:

This create a combined library with the ASI SystemC simulator, and with the UVM-SC framework and adapter libraries (that is, the ASI SystemC flavors of the UVM-SC framework and the UVM-ML adapter for UVM-SC).

```
% g++ -g -fPIC –m32 -o ./liball_osci_questa.so \
    -I$OSCI_INSTALL/include \
    -I$UVM_ML_HOME/ml/adapters/uvm_sc \
    -I$UVM_ML_HOME/ml/adapters/uvm_sc/osci \
    -I$UVM_ML_HOME/ml/frameworks/uvm/sc –I. \
    -shared -Xlinker -rpath \
    -Xlinker $UVM_ML_HOME/ml/libs/osci/2.3/4.4/ \
    -L$UVM_ML_HOME/ml/libs/osci/2.3/4.4/ \
    -luvm_sc_fw_osci -luvm_sc_ml_osci sc.cpp
```

> The directory paths /2.3/4.4/ represent /<OSCI version>/<g++ version>. You can use any of the supported ASI SystemC library versions (for example, 2.2, 2.3, as indicated by the libs/osci/* sub-directories available in the installation). The <g++ version> should be one of the supported compiler versions (for example, 4.1 or 4.4), and the same version used when installing UVM-ML.

Inputs and flags: Most of the lines above are generic for ASI SystemC compilation. However, the lines in blue text relate specifically to running UVM-ML, and have the following significance:

Flag	Description
`-I$UVM_ML…/ml/adapters/…`	include directories, where the UVM-ML SC adapter code is
`-I$UVM…/ml /frameworks/…`	include directories, where the UVM-ML SC framework code is
$UVM_ML…/ml/libs/osci/…	The location of the relevant shared objects and link files (stated in two lines, see both the –Xliner and the –L options)
`-luvm_sc_fw_osci`	The shared object of the uvm-sc framework (in the –L location)
`-luvm_sc_ml_osci`	The shared object of ASI SystemC lib and the uvm-sc adapter (in –L loc.)
`sc.cpp`	The user C++ code file(s), in this simple example, just one file

Output: The output is the shared object to be used in the next step: liball_osci_questa.so

5. Finally Questa is launched

```
% vsim -c -do 'run -all; exit -f' \
```

```
-sv_lib ./libuvm_dpi \
-gblso $UVM_ML_HOME/ml/libs/backplane/4.4/libuvm_ml_bp.so \
-sv_lib $UVM_ML_HOME/ml/libs/uvm_sv/4.4/libuvm_sv_ml \
-gblso $UVM_ML_HOME/ml/libs/uvm_sv/4.4/libuvm_sv_ml.so \
-gblso ./liball_osci_questa.so \
-sv_lib $UVM_ML_HOME/ml/libs/osci/2.3/4.4/libuvm_sc_fw_osci \
-sv_lib ${UVM_ML_HOME}/ml/libs/osci/2.3/4.4/libuvm_sc_ml_osci topmodule \
-dpioutoftheblue 1
```

Inputs and flags : Most of the lines above are generic for Questa and UVM SystemVerilog. However, the lines in blue text relate specifically to running UVM-ML, and have the following significance:

Flag	Description
`./libuvm_dpi`	DPI shared objected, prepared in step 1
`$UVM_ML…/libuvm_ml_bp.so`	The backplane shared library (from ML install)
`$UVM_ML…/libuvm_sv_ml.so`	The DPI-C code of the SV adapter (from ML install)
liball_osci_questa.so	The shared object created in step 1 (ASI SystemC + SC ML libs)
-dpioutoftheblue 1	Enable calling SystemVerilog DPI export functions from ASI SystemC or UVM-ML backplane to enable automated synchronization between the simulator and the SystemC framework

The directory paths /4.4/ r epresent the <g++ version> version used. You can use any of the supported compiler versions (for example, 4.1 or 4.4), and this should be the same version used when installing UVM-ML.

The directory paths /2.3/4.4/ represent /<OSCI version>/<g++ version>. You can use any of the supported ASI SystemC library versions (for example, 2.2, 2.3, as indicated by the libs/osci/* sub-directories available in the installation). The <g++ version> should be one of the supported compiler versions (for example, 4.1 or 4.4), and the same version used when installing UVM-ML.

Running Questa with the SystemC Portable Adapter

For using the native Questa SystemC, the UVM-ML portable adapter needs to be linked in. To use the portable adapter with Questa with any of the SystemC examples, choose a fitting examples directory and invoke make with the questa_sc target. For example:

```
% make -f $UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv/Makefile questa_sc
```

The main commands that are involved can be seen if you issue make with its --dry-run option (the outcome below was slightly edited for clarity)

```
      % cd $UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv
      % make questa_sc --dry-run
      UVM_ML_TEST_RUN_DIR is $UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv
      make -C $UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv -f ./Makefile.questa
run_questa_sc
      # setting some environment variables
      UVM_HOME=${UVM_ML_HOME}/ml/frameworks/uvm/sv/1.1d-ml; export UVM_HOME
      # systemc compilation
      sccom -g -DUVM_ML_PORTABLE -DUVM_ML_PORTABLE_QUESTA -
I${UVM_ML_HOME}/ml/frameworks/uvm/sc/portable \
      -I${UVM_ML_HOME}/ml/frameworks/uvm/sc -I${UVM_ML_HOME}/ml/adapters/uvm_sc/common \
      -I${UVM_ML_HOME}/ml/adapters/uvm_sc -I${UVM_ML_HOME}/ml/adapters/uvm_sc/portable \
      -DMTI_BIND_SC_MEMBER_FUNCTION ./sc.cpp
      sccom -g -link -lib ${UVM_ML_HOME}/ml/portable_adapter_for_questa \
      -lib ${UVM_ML_HOME}/ml/boost_library_for_questa \
      -Wl,-rpath,$MTI_HOME/mti/10.3/modeltech/gcc-4.5.0-linux_x86_64/lib64
      # Questa compilation
      vlog -dpicppinstall ${UVM_ML_CC} -suppress 2218,2181 +define+USE_UVM_ML_RUN_TEST \
      +incdir+${UVM_ML_HOME}/ml/frameworks/uvm/sv/1.1d-ml/src \
      +incdir+${UVM_ML_HOME}/ml/frameworks/uvm/sv/1.1d-ml \
      +incdir+${UVM_ML_HOME}/ml/frameworks/uvm/sv/1.1d-ml/tlm2 \
      +incdir+${UVM_ML_HOME}/ml/frameworks/uvm/sv/1.1d-ml/base \
      +incdir+. +incdir+${UVM_ML_HOME}/ml/frameworks/uvm/sv/1.1d-ml/macros \
      +incdir+${UVM_ML_HOME}/ml/adapters/uvm_sv -sv +acc -l questa_comp.32.log \
      -timescale 1ns/1ns +define+UVM_ML_PORTABLE +define+UVM_ML_PORTABLE_QUESTA \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sv/1.1d-ml/src/uvm.sv \
      ${UVM_ML_HOME}/ml/adapters/uvm_sv/uvm_ml_adapter.sv ./test.sv
      # Questa Run
      vsim -c -do 'run -all; exit -f' -sv_lib
$UVM_ML_HOME/ml/examples/use_cases/side_by_side/sc_sv/libuvm_dpi \
      -gblso ${UVM_ML_HOME}/ml/libs/backplane/4.4/libuvm_ml_bp.so \
      -sv_lib ${UVM_ML_HOME}/ml/libs/uvm_sv/4.4/libuvm_sv_ml \
      -gblso ${UVM_ML_HOME}/ml/libs/uvm_sv/4.4/libuvm_sv_ml.so \
      -novopt -l questa_sc_proc.32.log topmodule sc_main
```

Alternatively, you can do it manually according to the steps described in the following two sub sections.

The portable adapter has a few limitations: it does not support global timeout, and unified hierarchy works only with SystemC on top of ***e*** or SystemVerilog.

General Instructions About the Portable Adapter

The portable UVM-SC adapter enables the use of UVM-ML with the native SystemC in Questa.

1. Though the SystemC modules and components will be instantiated in a vendor SC simulator statically, the topmost SC testbench component must be also explicitly specified as an argument of uvm_ml_run_test() (see example below). The portable SC adapter does not attempt to instantiate the top component, specified as an argument, but rather searches for it using the standard function sc_find_object(). This allows UVM-ML to recognize, for example, port names as belonging to SC.

2. The hierarchical name of a static SystemC module or uvm_component, instantiated at sc_main, will begin with sc_main. This is correct for uvm_ml_run_test() as well. For example:

```
// SC:
int sc_main(int argc, char** argv) {
#ifdef UVM_ML_PORTABLE_QUESTA
sctop a_top("sctop");
sc_start(-1);
#endif
return 0;
};
```

```
// SV:
initial begin
string tops[2];
tops[0] = "SV:test";
`ifdef UVM_ML_PORTABLE_QUESTA
tops[1] = "SC:sc_main/sctop";
`else
tops[1] = "SC:sctop";
`endif
uvm_ml_run_test(tops, "");
end
```

3. Questa uses the "/" separator as the hierarchical separator. Since the portable adapter supports static SC hierarchy, the hierarchical SystemC names shall use the "/" separator rather than ".". For example:

```
// SV:
res = uvm_ml::connect("sc_main/sctop/producer/aport",sub1.aimp.get_full_name());
```

4. The tests, currently working with the quasi-static SC hierarchy, need to be carefully modified. In particular, all the child component/module allocations (if any) must be moved (under #ifdef) to the constructor. The same is true for registration of the ML-connected TLM ports and sockets (because uvm_ml_register() allocates a proxy port/export under the hood). For example:

```
// SC:
class env : public uvm_component {
env(sc_module_name name) : uvm_component(name), phase_count(0), bphase_started(false),
bphase_ended(false), bphase_ready_to_end(false), btest_passed(true), bsctop_passed(true)
{
#ifdef MTI_SYSTEMC
prod = new producer<packet>("producer");
uvm_ml_register(&(prod->aport));
#endif
}
...
};
```

5. Another, sometimes necessary modification is to move the native SC-SC TLM port connections from the connect_phase() callbacks (if any) to before_end_of_elaboration(). For example:

```
void before_end_of_elaboration(){
#ifdef MTI_SYSTEMC
prod->aport(sub->aexport);
#endif
}
void connect_phase(uvm_phase *phase){
#ifndef MTI_SYSTEMC
prod->aport(sub->aexport);
#endif
}
```

6. In order to use the unified ML hierarchy (if you decide to) you need also to make some code modifications – please refer to the example in the test/hierarchy directory.

Manual Usage of the Portable Adapter

In order to use the UVM-SC portable adapter with Questa, these steps should be followed:

1. First compile the portable adapter and the boost library (two compilation steps)

```
      % sccom -g -64 -work <portable_adapter_location> -DUVM_ML_PORTABLE -
DUVM_ML_PORTABLE_QUESTA \
      -iquote${UVM_ML_HOME}/ml/frameworks/uvm/sc/ \
      -iquote${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/include \
      -iquote${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages \
      -I${UVM_ML_HOME}/ml/frameworks/uvm/sc/base -I${UVM_ML_HOME}/ml/frameworks/uvm/sc \
      -I${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/portable \
      -I${UVM_ML_HOME}/ml/backplane \
      -I${UVM_ML_HOME}/ml/frameworks/uvm/sc/portable \
      -I${UVM_ML_HOME}/ml/frameworks/uvm/sc/base \
      -I${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/portable \
      -I${UVM_ML_HOME}/ml/frameworks/uvm/sc -I${UVM_ML_HOME}/ml/adapters/uvm_sc/ \
      -I${UVM_ML_HOME}/ml/adapters/uvm_sc/portable \
      -I${UVM_ML_HOME}/ml/backplane \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_component.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_factory.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_globals.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_ids.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_manager.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_object.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_packer.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_typed.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_barrier.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_phase.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_common_phase.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_schedule.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_common_schedule.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_resource_base.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/base/uvm_resource_pool.cpp \
      ${UVM_ML_HOME}/ml/adapters/uvm_sc/common/ml_tlm2/ml_tlm2.cpp \
      ${UVM_ML_HOME}/ml/adapters/uvm_sc/common/uvm_ml_adapter.cpp \
      ${UVM_ML_HOME}/ml/adapters/uvm_sc/common/uvm_ml_config_rsrc.cpp \
      ${UVM_ML_HOME}/ml/adapters/uvm_sc/common/uvm_ml_hierarchy.cpp \
      ${UVM_ML_HOME}/ml/adapters/uvm_sc/common/uvm_ml_packer.cpp \
      ${UVM_ML_HOME}/ml/adapters/uvm_sc/common/uvm_ml_phase.cpp
      % sccom -g -64 -work <boost library location> \
      -I${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost \
      -I${UVM_ML_HOME}/ml/frameworks/uvm/sc \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/c_regex_traits.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/cpp_regex_traits.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/cregex.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/fileiter.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/icu.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/instances.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/posix_api.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/regex.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/regex_debug.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/regex_raw_buffer.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/regex_traits_defaults.cpp
\
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/static_mutex.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/usinstances.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/w32_regex_traits.cpp \
      ${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/wc_regex_traits.cpp \
```

```
${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/wide_posix_api.cpp \
${UVM_ML_HOME}/ml/frameworks/uvm/sc/packages/boost/libs/regex/src/winstances.cpp
```

2. Prepare the DPI library

```
% g++ -m64 -g -fPIC \
–I$MTI_HOME/include \
-shared -o libuvm_dpi.64.so \
$UVM_HOME/src/dpi/uvm_dpi.cc
```

3. Next compile and link the SystemC code with the portable adapter

```
% vlib work
% sccom -g -64 -DUVM_ML_PORTABLE -DUVM_ML_PORTABLE_QUESTA \
–I$UVM_ML_HOME/ml/frameworks/uvm/sc/portable \
-I$UVM_ML_HOME/ml/frameworks/uvm/sc \
-I$UVM_ML_HOME/ml/adapters/uvm_sc/common \
-I$UVM_ML_HOME/ml/adapters/uvm_sc \
-I$UVM_ML_HOME/ml/adapters/uvm_sc/portable \
-DMTI_BIND_SC_MEMBER_FUNCTION -64 sc.cpp
sccom -g -64 -link \
-lib <framework and adapter location> \
-lib <boost library location>
```

4. Then compile the SystemVerilog code

```
% vlog -dpicppinstall g++ -64 -suppress 2218,2181 \
+incdir$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/src \
+incdir$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml \
+incdir$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/tlm2 \
+incdir$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/base \
+incdir. \
+incdir$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/macros \
+incdir$UVM_ML_HOME/ml/adapters/uvm_sv \
-sv +acc -timescale 1ns/1ns \
+define+UVM_ML_PORTABLE +define+UVM_ML_PORTABLE_QUESTA \
$UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/src/uvm.sv \
$UVM_ML_HOME/ml/adapters/uvm_sv/uvm_ml_adapter.sv test.sv
```

5. Finally run the Questa simulator as follows

```
% vsim -c -do 'run -all; exit -f' -64 \
-sv_lib ./libuvm_dpi.64 \
-gblso $UVM_ML_HOME/ml/libs/backplane/4.5/64bit/libuvm_ml_bp.so \
-sv_lib $UVM_ML_HOME/ml/libs/uvm_sv/4.5/64bit/libuvm_sv_ml \
-gblso $UVM_ML_HOME/ml/libs/uvm_sv/4.5/64bit/libuvm_sv_ml.so \
-novopt topmodule sc_main
```

The examples make use of specific version:

- 14.1 is the IES version. It can be replaced with $IES_VERSION or any supported IES version.
- 4.4 is the gcc version. It can be replaced with $UVM_ML_COMPILER_VERSION or any supported gcc version.
- 2.3 is the ASI SystemC version. It can be replaced by any of the supported versions.

The Underlying Infrastructure: UVM-ML Make Files and Libraries

The UVM-ML release package contains Makefiles that embody the compile and run processes. These in turn make use of several libraries created during the installation. The following subsections describe those Makefiles and Libraries.

This information is provided here for advanced users who want to understand how things work, or make use of this infrastructure. Users who just want to learn how to run UVM-ML can skip this.

Section Contents

UVM-ML Makefiles

The UVM-ML release package contains generic Makefiles for IES, VCS and Questa, all located in directory `$UVM_ML_HOME/ml/tests/` , named `Makefile.` *<simulator>* . To use them in environments you create or modify, you should provide some parameters, through local Makefiles. Follow the examples in any of the `$UVM_ML_HOME/ml/examples/...` directories (there too you will find local make files, named `Makefile.ies` , `Makefile.vcs` , or `Makefile.questa`).

The provided Makefiles have multiple targets:

- make ies – to run with IES when the top level components are specified procedurally in uvm_ml_run_test()
- make ies_ncsc_cl – to run with IES when the top components are declared on the command line (the command line option works on IES only)
- make ies_osci – to run with IES and use the ASI SystemC simulator for the SystemC code
- make ies_osci_cl – to run with IES and use the ASI SystemC simulator for the SystemC code; the top components are declared on the command line
- make ies_osci_proc – to run with IES and use the ASI SystemC simulator for the SystemC code; the top components are specified procedurally in the uvm_run_test() command
- ies_ncsc_cl_multistep – to run with IES in multi-step mode when the top components are declared on the command line
- ies_ncsc_proc_multistep – to run with IES in multi-step mode when the top components are specified procedurally in the uvm_run_test() command
- ies_osci_cl_multistep – to run with IES in multi-step mode when the top components are declared on the command line; use the ASI SystemC simulator for the SystemC code
- ies_osci_proc_multistep – to run with IES in multi-step mode when the top level components are specified procedurally in uvm_ml_run_test(); use the ASI SystemC simulator for the SystemC code
- make vcs – to run with VCS and use the ASI SystemC simulator for the SystemC code
- make questa – to run with Questa and use the ASI SystemC simulator for the SystemC code
- make questa_sc – to run with Questa and use the native Questa simulator for the SystemC code
- make clean – to remove the results of the simulation except for the log files
- make distclean – to remove the results of the simulation including the log files

Several controls are available to modify the behavior of the makefiles:

- GUI_OPT="-gui" to add the -gui option to irun
- EXIT_OPT="" to turn off the -exit option of irun

- EXTRA_IRUN_ARGS use to add argument to irun (for example, setenv EXTRA_IRUN_ARGS "-enable_DAC")
- EXTRA_IRUN_COMPILE_ARGS use to pass any custom arguments to irun. Applicable to all multi-step targets
- EXTRA_NCSIM_ARGS use to pass any custom arguments to ncsim. Applicable to all multi-step targets
- EXTRA_OSCI_ARGS use to add arguments to ASI SystemC simulator compilation
- EXTRA_SN_COMPILE_ARGS use for Makefile.vcs / Makefile.questa
- EXTRA_VCS_ARGS use for Makefile.vcs
- EXTRA_VSIM_ARGS use for Makefile.questa

UVM-ML Libraries

UVM-ML makes use of several libraries created during the installation. They are used in the various flows as shown below. This section gives a short description of their location and purpose.

> There are separate versions of the libraries according to the tool version and the compiler version, as well as separate libraries for 32-bit and 64-bit platforms.
> All locations are subdirectories under `$UVM_ML_HOME/ml/libs/`

Library	Name	Location	Comments
backplane	libuvm_ml_bp.so	backplane/<compiler-version>[/64bit]	The backplane shared library (used with IES, VCS, Questa).
backplane auxiliary	libml_uvm.so	backplane/<IES-version>/ <compiler-version> [/64bit]	An Incisive-only auxiliary and auto-loader for libuvm_ml_bp.so (used for both NCSC and ASI -SC)
DPI-C of SV adapter	libuvm_sv_ml.so	uvm_sv/<compiler-version>[/64bit]	DPI-C code of the SV adapter (used with IES, VCS, Questa).

Specman with UVM-e adapter	libsn_sn_uvm_ml.so	uvm_e/ies_<IES-version>/<compiler-version>[/64bit]	Specman library containing the UVM-ML Specman adapter code (used with IES, VCS, Questa).
Specman with UVM-e adapter	sn_uvm_ml	uvm_e/ies_<IES-version>/<compiler-version>[/64bit]	Executable used by sn_compile.sh when you are compiling e files on top of the adapter
UVM-SC adapter for NCSC	libuvm_sc_ml.so	ncsc/<IES-version>/<compiler-version>[/64bit]	UVM-SC adapter for NCSC (used for NCSC runs only)
UVM-SC framework for NCSC	libuvm_sc_fw.so	ncsc/<IES-version>/<compiler-version>[/64bit]	UVM-SC framework for NCSC (used for NCSC runs only)
UVM-SC adapter+fw for NCSC	libuvm.so	ncsc/<IES-version>/<compiler-version>[/64bit]	an NCSC-only bundle containing UVM-SC adapter and framework (used for NCSC runs only)
UVM-SC adapter for ASI SystemC	libuvm_sc_ml_osci.so	osci/<osci-version>/<compiler-version>[/64bit]	Includes the uvm-sc adapter compiled for ASI-SC , and an ML enabled version of the ASI-SC library (used with IES, VCS, Questa).
UVM-SC framework for ASI SystemC	libuvm_sc_fw_osci.so	osci/<osci-version>/<compiler-version>[/64bit]	UVM-SC framework for ASI-SC (used with IES, VCS, Questa).

- The <osci-version> above should be one of the supported ASI SystemC library versions (for example, 2.2, 2.3). The supported version are indicated by the libs/osci/* sub-directories available in the installation.
- The <compiler-version> should be one of the supported g++ compiler versions (for example, 4.1 or 4.4).
- The <IES-version> should be one of the supported IES versions (for example, 13.2, 14.1).
- the optional /64bit should be added to the path in case compiling and running on a 64 bit platform.

ASI

10

Debugging in a Multi-Language Environment

Debugging includes several tasks:

- Viewing static information:
  - View the components and their status.
  - View history--transactions that were recorded during the test.
- Viewing dynamic information:
  - Trace sequences, objections, testflow.
  - Report messages from verification components.
- Code debugging.

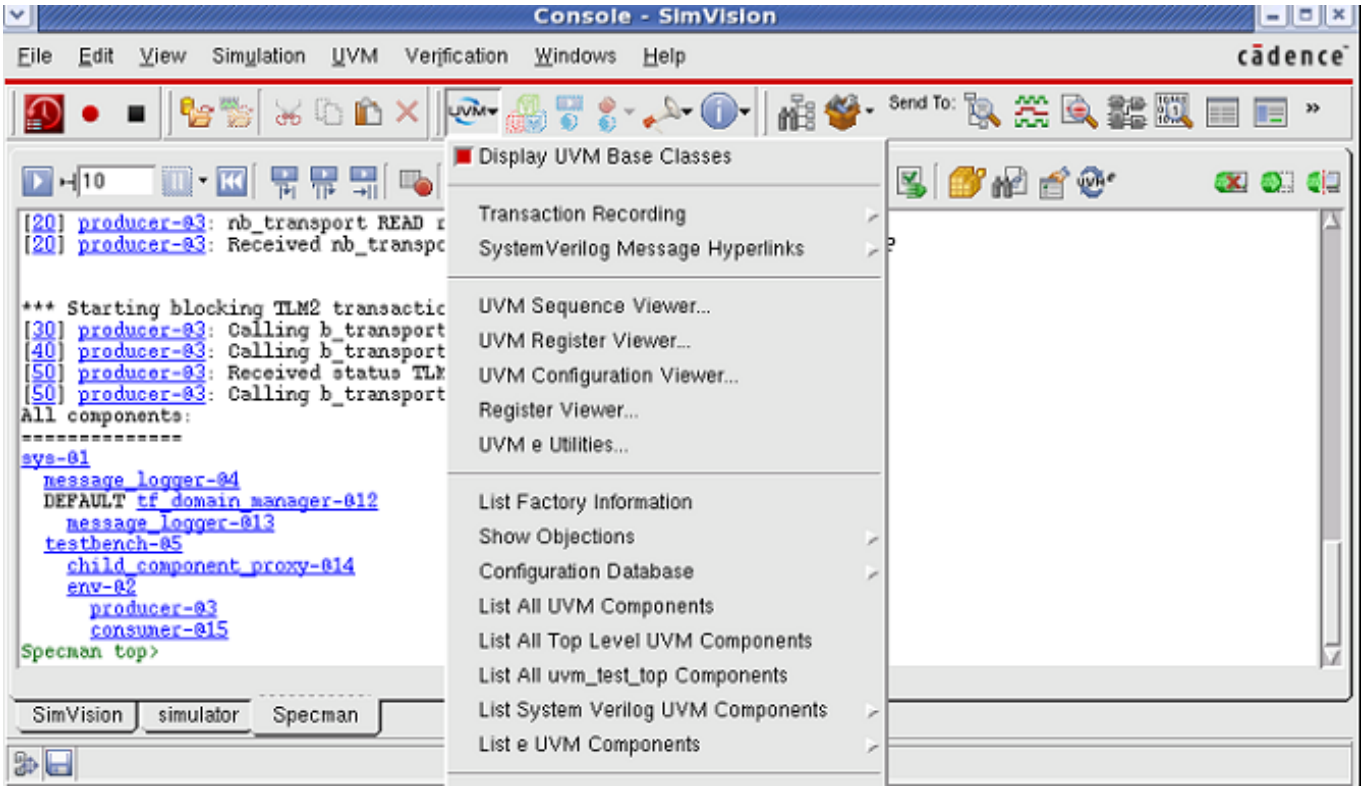
Some of these tasks can be performed in "multi-language mode". For other tasks, the multi-language mode is still under development.

In This Section:

Data Browsing in a Multi-Language Environment

If you are using IES and SimVision, there are several tools for browsing the verification environment data:

- SimVision's UVM tools provide information about ***e*** and SystemVerilog UVM components.
- The command **uvm_ml print_tree** presents the full hierarchy-- ***e*** units, SystemVerilog UVM components and SystemC UVM components--in ASCII form.
- The command **uvm_components -list** presents SystemVerilog components, ports, and sockets.



uvm_ml print_tree Output

```
sv:    uvm_test_top [svtop]
sv:    \__tb [testbench]
e:        \__xbus_uvc [MY_XBUS'bus_name xbus_env_u]
e:           |__synch [TRUE'has_checks MY_XBUS'bus_name xbus_synchronizer_u]
e:           |__smp [xbus_signal_map_u]
e:           |__bus_monitor  [TRUE'has_checks xbus_bus_monitor_u]
e:           |__active_masters[0] [MASTER'kind ACTIVE'active_passive xbus_agent_u]
```

Note that you can always browse the data of each of the hierarchies separately.

Notes:

- SystemVerilog components are built and connected in the **build** and **connect** phases, so you can browse the environment after end of elaboration.
    - You can call the **print_topology()** task.
- if you issue Specman command **print sys,** you will not see the units instantiated under SystemVerilog or SystemC.
    - You can use the UVM Utility window or the **print units** command.

See also

- "Incisive Debugging Commands for Use in a UVM-ML Environment" in the *UVM-ML OA Reference*

Debugging Configuration

Errors in configuration occur primarily due to these reasons:

- **uvm_config_<>::get()/uvm_config_get()** is not called.
- Misspellings in the parameters passed to set/get.
    - A simple typing error
    - A wrong path (for example, "env.agents[0]" instead of "sys.env.agents[0]")

Because the configuration propagation mechanism is based on a database of strings, misspellings are not reported.

Currently, there is no a multi-language debugger for configuration, so you should trace whether configuration took place as planned in SystemVerilog and in ***e,*** independently.

Getting information from the UVM-SystemVerilog configuration dataBase

Using SimVision, you can trace configuration with the UVM Debugger: UVM->Configuration Database->Trace->ON

You can also use the following SystemVerilog args:

- UVM_CONFIG_DB_TRACE
- UVM_RESOURCE_DB_TRACE

Getting information about e configuration

Because configuration values retrieved by ***e*** unit is used within constraints, you can use the Specman **trace gen** command to see how structs are generated.

Debugging Data Transfer on Ports

Data transfers via ports can require debugging either because data written to a port is not received or the data received is corrupted.

For debugging such cases, you can take these actions:

1. View all the connected ports/sockets in the environment. See if the ports you debug are connected.
2. Trace the activity on the port to see when data is written on the ports.
3. Debug the execution of the port implementation.

Commands listing all the ports

There are several IES Tcl commands that list all the ports in the environment:

- **uvm_ml print_tree -ports**
- **uvm_components -list**

Command for listing multi-language connections

To see the connections between ports and sockets in a multi-language environment, use the IES Tcl **uvm_ml print_connections** command after the connect phase. This command lists all successful connections created during the construction of the verification environment.

Output of ML connections

```
ncsim> uvm_ml print_connections -type
List of UVM-ML connections:
Specman:sys.e_env.producer.nb_socket [tlm_initiator_socket of tlm_generic_payload]
-> UVM SV:uvm_test_top.sv_env.consumer_1.nb_target_socket [uvm_tlm_nb_target_socket]
Specman:sys.e_env.producer.b_socket [tlm_initiator_socket of tlm_generic_payload]
-> UVM SV:uvm_test_top.sv_env.consumer_1.b_target_socket [uvm_tlm_b_target_socket]
```

Tracing port activity

A typical debugging scenario: You want to track a data item progress in the verification environment. Typically, the item is created by a bus collector and passed to the bus monitor, and from the monitor to a checker and perhaps also to a scoreboard.

When the interface between components is using ports, there are various ways to trace the data item flow throughout the verification environment components.

Commands for tracing activities on ports

There are several **trace** commands, providing different levels of details.

- Specman commands:
  - **trace ml_ser**
    - Prints a message and the item's fields, whenever an item is written written on a port

Print outs of trace ml_ser

```
      [500] XBUS_IN_ML: (info - serializing) method 'write' started serialization of 'value' :
'xbus_trans_s' = MONITOR xbus_trans_s-@14
      object 'MONITOR xbus_trans_s-@14' : 'xbus_trans_s'
      field 'kind' : 'xbus_trans_kind_t' = MONITOR
      field 'addr' : 'xbus_addr_t' = 0x0044
      field 'size_ctrl' : 'uint (bits: 2)' = 0x2
      field 'read_write' : 'xbus_read_write_t' = WRITE
      field 'size' : 'uint[1..2,4,8]' = 0x4
      field 'data' : 'list of byte' = 0x4 elements
      data[0] : 'byte' = 0xbe
      data[1] : 'byte' = 0x31
      data[2] : 'byte' = 0xcc
      data[3] : 'byte' = 0x25
```

Source debugging

If viewing the items does not give you enough information and you want to debug your source code, you have to set two break points: the first on the line writing the port and another one on the implementation of the port.

Notes:

- For setting break points and source debugging the SystemVerilog code with Simvision, you should pass the **-linedebug** flag to irun.

See Also:

- "trace esi" in *Specman Command Reference*
- "Specman Debugging Commands for Use in a UVM-ML Environment" in *UVM-ML OA Reference*
- "Incisive Debugging Commands for Use in a UVM-ML Environment" in the *UVM-ML OA Reference*

11

Troubleshooting

This section contains troubleshootnig tips for common problems that occur during creation of multi-language verification environments.

In This Section

Install Troubleshooting

Following is an issue that you might encounter while installing UVM-ML.

Issue	Probable Solution

Install fails with error messages from sc_utils_ids.cpp, such as:
`/sc_utils_ids.cpp:110: error: 'getenv' is not a member of 'std'`
	This is a known issue with the Accellera SystemC (open source) version systemc-2.2.0 when compiled with GNU C++ version above 4.3. See http://openesl.org/systemc-wiki/Installation for more information.
	The suggested solution is to use the patch in the link above or manually modify sysc/utils/sc_utils_ids.cpp so that it explicitly includes the following files:
	`#include "string.h"`
	`#include "cstdlib"`

Compilation / Build Troubleshooting

This section describes the issues most commonly encountered while compiling or building multi-language environments using UVM-ML.

General recommendations to follow when your environment fails to compile:

- Clean up the environment. Either use **make clean** or remove directories created by the tools (for example, INCA_libs in IES).
- Check that environment variables are set correctly.
  - Check the environment variables: Run **$UVM_ML_HOME/ml/test_env.csh** .
  - Set the environment variables: Run **$UVM_ML_HOME/ml/setup_<32/64>.csh** .

Error Encountered	Possible Solution
Verilog compilation reports multiple uvm_pkg's. For example, when using Incisive: `ncvlog: *E,MULTPK (...): Multiple (2) packages named "uvm_pkg" were found in the searched libraries`	If different parts of the code are compiled with different versions of UVM-SystemVerilog, a conflict is detected by the Verilog compiler. This could result from creating a snapshot with the default version in the IES release and then adding on top the UVM-ML OA version. Make sure all compilations refer to same location--the UVM-ML install.

The simulator reports that no components are instantiated: `UVM_FATAL @ 0: reporter [NOCOMP] No components instantiated. You must either instantiate at least one component before calling run_test or use run_test to do so. To run a test using run_test, use +UVM_TESTNAME or supply the test name in the argument to run_test(). Exiting simulation UVM_FATAL @ 0: reporter [NOCOMP] No components instantiated. You must either instantiate at least one component before calling run_test or use run_test to do so. To run a test using run_test, use +UVM_TESTNAME or supply the test name in the argument to run_test(). Exiting simulation`	The cause for this may be that the user's UVM-SystemVerilog code includes a call to **run_test()** despite the fact that this is a UVM-ML OA environment in which only **uvm_ml_run_test()** should be called. In this case, there is a conflict between UVM-SystemVerilog and UVM-ML, both attempting to drive the simulation. In UVM-ML verification environments, use **uvm_ml_run_test()** only and remove all calls to **run_test()** .
The Specman library fails to load: `rundynlib - failed to load the provided library .../libsn_sn_uvm_ml.so`	This failure is caused by a difference between the currently used Specman version and the one used in installing UVM-ML OA. Failing to load Specman library. `rundynlib - failed to load the provided library .../libsn_sn_uvm_ml.so` UVM-ML OA must be rebuilt with the current Specman version. Use the **- clean** flag with the **install.csh** script.

"UVM-ML BP_ERROR >> No framework with name 'e' has been registered"	If the DUT is in VHDL, old Specman versions (prior to IES 13.10 S025, 13.20 S012 or 14.10 S002) have a bug that causes 'e' not to register in the UVM-ML frameworks list. Either migrate to a newer Specman version or call the command **set uvm_ml** before running the simulation.
You encounter an error message like one of the following: • NCVLOG cannot be a slave adapter • The declarations in your currently loaded e modules do not match the declarations in the e modules that were available in the … stub file creation	Some ***e*** ports connected to a DUT require generation of auxiliary code in a file commonly named a "stub file". Generation of the stub file happens when the unified multi-language testbench hierarchy is not yet known. Hence, the ***e*** unit and port attributes of the units instantiated under SystemVerilog or SystemCare not taken into account at stub-generation time. See Stub Generation for an e Sub-Tree .
ncvlog: *E,NOPBIND (./test.sv,22|12): Package uvm_ml could not be bound.	This error message is issued when the UVM_HOME environment variable is set to a path that does not contain the multi-language-ready version of UVM-SystemVerilog. Unset the UVM_HOME environment variable: % unsetenv UVM_HOME

In This Section:

For more compilation issues, see:

Compilation With IES

Following are issues commonly encountered while compiling an environment using IES:

Error Message	Probable Solution
`ncvlog: *E,NOPBIND (./ test.sv ,22|12): Package uvm_ml could not be bound`	For UVM-ML, you must use the multi-language ready UVM-SystemVerilog library included in the release. This error message is issued when the UVM_HOME environment variable is set to a path that does not contain the multi-language ready version of UVM-SystemVerilog. Unset the UVM_HOME environment variable : `unsetenv UVM_HOME`
`ncelab: *F,SCILDD: Could not load SystemC model library...`	irun must pass the appropriate arguments to the linker, as shown in irun_uvm_ml.64.f. `irun -f $UVM_ML_HOME/ml/tests/irun_uvm_ml.64.f ....`

Compilation With Questa

This information is yet to be provided.

See Also

- Running UVM-ML with Questa

Compilation With VCS

This informationn is yet to be provided.

See Also

- Running UVM-ML with VCS

Missing End of Test Activities

A test using UVM-SystemVerilog ends with a UVM Report Summary, listing the number of errors issued during the test. If your environment contains UVM-SystemVerilog code and there is no such a summary at the end of the test, the SystemVerilog parts of the verification environment did not execute the post-run phases, namely the **check_phase** .

This can happen if the test is stopped by an ***e*** component calling **stop_run()** . This ***e*** routing stops the test without calling UVM-SystemVerilog post run phases.

For information about stopping the test in multi-language environments, see Ending (Stopping) The Test

Handling Failures in Passing Items via Ports

This section describes failures you might encounter when passing items via ports over a language barrier.

Failure in Connecting Ports

Port connections are based on strings--the full port paths. If a string passed to the connect method is not accurate, the ports will not be connected, but there are no warnings during compilation. During the run, when the port is being written, the test will fail with an error message similar to the following:

```
ncsim: *F,MLUVM06: Could not find an adapter for originating language for port/export name ...
```

If you are not sure what the exact path to the port is, you can print it using ***e*** 's **e_path()** and SystemVerilog's **get_full_name()** .

Printing the full path to a port

```
unit consumer {
b_tsocket  : tlm_target_socket of tlm_generic_payload is instance;
post_generate() is also {
out("path to the e port ", b_tsocket.e_path());
};
};

class producer extends uvm_component;
uvm_tlm_b_initiator_socket #() b_initiator_socket;
function void build_phase(uvm_phase phase);
super.build_phase(phase);
b_initiator_socket  = new("b_initiator_socket", this);
$display("path to the SV port %s",b_initiator_socket.get_full_name());
endfunction
endclass
```

Fatal Error in Unpacking

This error occurs when the consumer--the component implementing the in port--fails to unpack the data. When passing data from ***e*** to SystemVerilog, the error message looks something like this:

```
UVM_ERROR @ 500: reporter [MLLIBPCKSZ] 1 ints needed to unpack integral, yet only 0
available.
UVM_ERROR @ 500: reporter [MLLIBPCKSZ] 1 ints needed to unpack integral, yet only 0
available.
UVM_FATAL @ 500: reporter [UNPKSZ] Unpacked fewer ints than were packed. This may be
due to a mismatch in class definitions across languages. Base class = xbus_trans_s.
```

This error message is telling you that it tries to unpack the list of bytes it got into the port data item and fails. The list of byte is of wrong length.

There can be two typical causes for such failures:

1. A mismatch in the types.
2. A bug in serialisation.

The following describes the steps for debugging such failures:

Check the types definition

Check that the type definitions for both data items--in ***e*** and in SystemVerilog--are identical. Even if you used mltypemap to create the items, errors can occur, such as someone modifying the files and changing the definition of one of the items.

Edit the SystemVerilog file containing the definition of the data item. For viewing the definition of the ***e*** type, it is recommended that you load the environment and issue the Specman **source** command. Example:

```
specman> source xbus_trans_s.*
```

Check the serializer

Both components--the ***e*** and the SystemVerilog component--have functions controlling the packing/unpacking of the items passed on the ports. In ***e*** , the method is defined in **uvm_ml_config** and called **tlm_pass_field()** . In SystemVerilog, the serialisation functions are implemented in a class extending **uvm_ml::uvm_ml_class_serializer** , and called **serialize()** and **deserialize()** .

If you used mltypemap, it created these functions in files named *_ser.e/sv.

- Make sure these files are included in the environment.
- Make sure that all serialisation functions (e / SV / SC) handle the same fields.

See also:

- Type Mapping

Testbench Configuration Troubleshooting

Unable to configure a SystemVerilog testbench top component

Problem: Configuration settings made in an initial block do not affect the build_phase of the testbench component if the test is activated automatically by IES (using **-uvmtest** / **-uvmtop** command line switches).

Explanation :

IES supports an automated activation of the test phases if either **-uvmtop** or **-uvmtest** is used with the **irun** command. An advantage of this method is that it eliminates race conditions between the testbench phases and SystemVerilog initial blocks. The automated test phasing begins after all global variable static initializers are executed and before any SystemVerilog processes, including initial blocks, start running.

The failure to configure the topmost UVM component appears because *uvm_config_<>* settings made in an initial block happen too late, after all the non-blocking pre-run phases, including *build* , are executed.

Solutions:

There are several possible options for solving this problem.

The most generic option is to use an explicit procedural activation of the test, rather than than an automatic one, meaning that you call the SystemVerilog task **uvm_ml_run_test** (), providing the test and top components as arguments, instead of using **irun** with **-uvmtest** / **-uvmtop** .

Example 1:

```
...

initial
uvm_config_db#(int)::set(null, "*", "agents_no", 1);

initial begin
    #0; uvm_ml_run_test("", "my_test");
end
```

The #0 delay allows the *uvm_config_db* settings made in an initial block to be executed before *uvm_ml_run_test()* .

Another option is to put all the topmost configuration settings in the UVM-SystemVerilog test's *build*

phase. This might be a preferable solution if you need to configure the testbench top, which is instantiated in the test component.

Example 2:

```
class my_test extends uvm_test;
      tb_env_t env;
   ...
      function void build_phase (uvm_phase phase);
          super.build_phase(phase);
          uvm_config_db#(int)::set(this, “env”, “agents_no”, 1);
          env = tb_env_t::type_id::create(“env”, this);
      endfunction
 endclass
```

The last option is to move the configuration settings from an initial block time to global initialization time. This can be achieved by writing a special initialization function and calling it especially as an initializer, as illustrated in the following example:

Example 3:

```
module dut_top;

     function bit initial_config_int (string field_name, int value);
          uvm_config_db#(int)::set (null, “*”, field_name, value);
      endfunction

     bit initial_config_agents_no = initial_config_int (”agents_no”,1);
     ...
 endmodule
```

Debug Troubleshooting

UVM-SystemVerilog uvm_phase debug command does not work as expected

Problem : The IES debug command **uvm_phase** does not execute as expected (for example, it does not stop at the requested phase).

Explanation : In UVM-SystemVerilog stand-alone, the phasing mechanism injects zero-time delays

(#0) between the phases. This allows execution of the debug commands between the phases. In a multi-language environment, this solution is not applicable because other frameworks may execute operations during such delta cycles which occur before the run phase.

UVM-ML OA executes the non-blocking phases without these zero-time delays. As a side effect, this breaks some debug commands like **uvm_phase** , which cannot stop at the expected time.

Solution : IES users should use the phase debug command **uvm_ml phase** which works fine in the UVM-ML OA environment.

Simulator Reset Troubleshooting

The following are issues that you could possibly encounter during a simulator reset

Issue	Solution

Currently, when you are working in GUI mode during a simulator reset, the GUI might issue spurious scope- or hierarchy-related warnings.	Examples of such warning are:
	`*W,SCDYUI: Quasi-static hierarchy may be inaccurate after a reset`
	`*W,SCPINV: Dynamic scope  no longer valid - traversing to last valid scope`
	Ignore all such warnings and continue normally with whatever you were doing before the warnings appeared.

Simulation crashes after the reset.	If any user C++ code in your environment calls pre-defined output stream methods during the static initialization stage--and that C++ code is not unloaded during simulator reset--it might cause a crash.
	
	Currently, any user C++ code that is not unloaded during simulator reset should not call the pre-defined output stream methods during the static initialization stage. For example, the following C++ code snippet will cause a post-reset crash:
	
	`struct s1 {`
	`s1() {`
	`cout << "S1 construction" << endl;`
	`}`
	`} v1;`
	
	The workaround is either to avoid calling predefined output stream methods during initialization or to incorporate the content of `$UVM_HOME/ml/testing/reset_crash_workaround.cpp` into your code:
	
	`volatile static struct s1 {`
	`streambuf *cout_buf;`
	`streambuf *cerr_buf;`
	`s1() {`
	`cout_buf = cout.rdbuf();`
	`cerr_buf = cerr.rdbuf();`
	`}`
	`~s1() {`
	`cout.rdbuf(cout_buf);`
	`cerr.rdbuf(cerr_buf);`
	`}`
	`} v1;`

UVM-ML OA debug command are no longer in effect after a simulator reset, particularly phase debug commands.	After the reset, reinvoke any debug commands that were active before the simulator reset.

See Also

- "IES Simulator Reset Command" in "UVM-ML Commands" in the *UVM-ML OA Reference*