# UVM-ML OA Reference

**Product Version 1.5.1**

**Aug 2015**

## Notice

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

# Contents

1

# UVM-ML Key Concepts

Verification projects often involve off-the-shelf verification components implemented in different languages (SystemVerilog, e, SystemC, C++) and based on different methodologies (flavors of UVM, VMM, different C++ class libraries). These components need to be reused with minimal changes. UVM-ML provides a way to do this.

The UVM-ML Open Architecture solution (UVM-ML) is based on and enhances UVM methodology:

- UVM-ML expands the UVM scope from a single language to multiple languages. There are no assumptions about the number or types of language and methodology frameworks to be integrated.
- UVM-ML is simulator and vendor independent and available to the community as open source.

⚠ In this Reference guide, "UVM-ML" is used to mean the UVM-ML Open Architecture solution. Note also that, in this guide, "UVM-ML" does *not* refer to the previous, Cadence-Specific UVM multi-language solution.

## In this Chapter

This chapter provides information that applies for all users of UVM-ML. Each of the following subsections points you to the relevant reference sections for more detailed information.

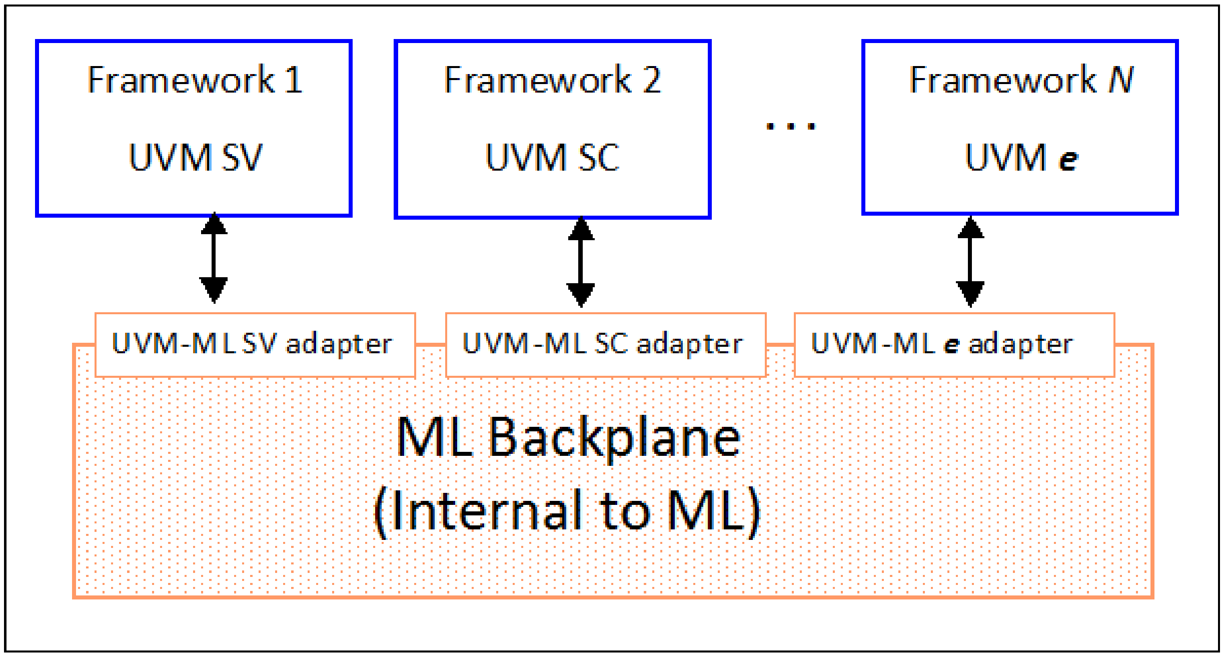
# Introduction to the UVM-ML Open Architecture Library

The primary elements of the UVM-ML open architecture library are the language frameworks, the framework adapters, and the multi-language backplane.

It is useful to understand these terms:

- A ***framework*** is a combination of language and methodology used for developing verification code (for example, UVM-SV, UVM-SC, UVM- ***e*** , ASI-SystemC, VMM). A new framework can be composed from a few simpler frameworks in the same language (for example, a combination of ASI-SystemC with a UVM SystemC library).
- The ***backplane*** is the internal central hub that connects the frameworks in a star topology (not peer-to-peer).
- A framework ***adapter*** is a bridge between the backplane and the user code in the framework--such that the framework is unaware of the backplane and the role it plays.

The following figure illustrates the basic architecture of frameworks, adapters, and backplane. Note that you can connect any number of frameworks from different vendors. This figure shows the frameworks UVM-SV, UVM-SC, and UVM-e, but your environment might include additional or different frameworks.

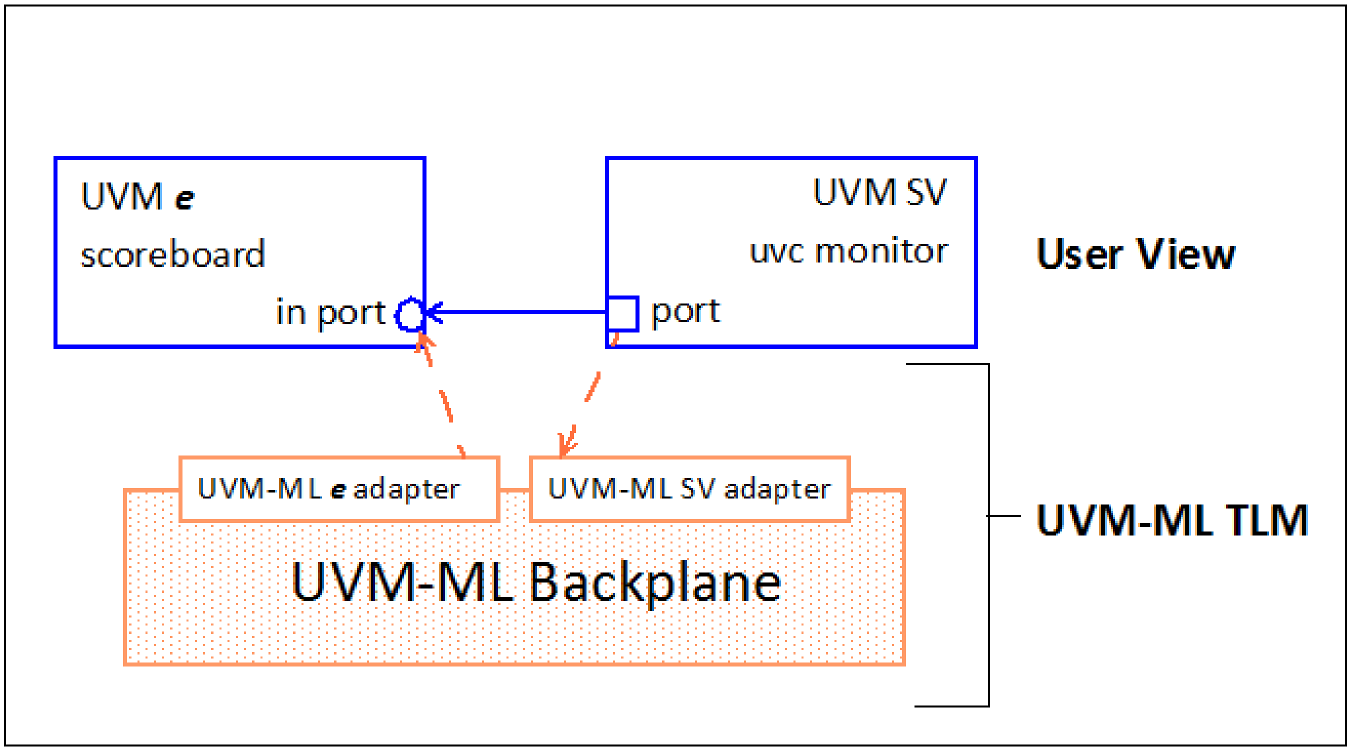


## The Work of the Backplane Is Transparent

The following figure, which represents a TLM1 transaction between the UVM-e framework and the UVM-SV framework, is another illustration of the architecture.

The emphasis in this figure is to show that the work done by the adapter and backplane are transparent to you, the user.

Notice that t he blue arrow shows the user's perspective of the communication path between the TLM ports, while the orange arrows show the actual communication path. The actual, "behind-the-scenes" communication path is not apparent to the user.



## The Frameworks Distributed with UVM-ML

The UVM-ML Open Source distribution provides two frameworks: UVM-SystemVerilog and UVM-SystemC. These frameworks are required to enable UVM-ML functionality:

- **UVM-SystemVerilog Framework:** This framework supports the standard UVM-SystemVerilog functionality, plus it includes several multi-language enablers that are currently not available in the standard library. UVM-ML supports two versions of UVM-SystemVerilog: 1.1d and 1.2. Note that UVM-SystemVerilog1.2 is currently supported as an early adopters version for Incisive and VCS only and is not fully validated yet.

- **UVM-SystemC Framework:** This framework is a standalone C++ class library that implements the UVM-SC methodology for use on top of a standard ASI-SystemC version or a vendor-proprietary SystemC implementation, such as Incisive.

  Note that the UVM-ML distribution also provides patches for ASI-SystemC versions 2.2 and 2.3 that make ASI-SystemC multi-language-ready. These patches are compiled on top of the ASI-SystemC installation during the UVM-ML installation and stored locally in the UVM-ML directories. (The original ASI-SystemC installation is not modified.)

## Adapters Provided with the Current Release

The adapters provided with the current release of UVM-ML are:

- UVM-ML SystemVerilog adapte r (simulator independent)
- UVM-ML SystemC adapter for Incisive
- UVM-ML SystemC adapter for the patched ASI-SystemC
- UVM-ML Portable adapter for the standard ASI-SystemC or vendor-specific SystemC simulator
- UVM-ML *e* adapter for Specman

Each adapter provides an API in its native language.The APIs include language constructs that are specific to UVM-ML. They are described in UVM-ML SystemVerilog Interface , UVM-ML SystemC Interface , and UVM-ML e Interface .

### Notes

- The portable adapter for SystemC is intended for use with various SystemC implementations. It can currently be used with the Questa simulator in combination with the UVM-SC framework provided with UVM-ML. (Questa-SystemC itself cannot be patched and does not contain the necessary multi-language enablers.)
- The UVM- ***e*** adapter works with most current Specman releases. For the specific range of releases that it works with, see < *UVM-ML_install_dir* >/ml/README_INSTALLATION.txt.

## See Also

- UVM-ML TLM Communication
- UVM-ML Hierarchy
- UVM-ML SystemVerilog Interface
- UVM-ML SystemC Interface
- UVM-ML e Interface

# Data Communication in a UVM-ML Environment

A multi-language solution like UVM-ML must provide a means for exchanging data between languages (or frameworks). This section describes how UVM-ML resolves this issue.

The information in this section applies for **hierarchical configuration** (described in UVM-ML Configuration in a Unified Hierarchy ) and **TLM communication** (described in UVM-ML TLM Communication ).

## In This Section

- The Types of Data Transferred Between UVM-ML Frameworks
- Understanding Class Serialization and De-Serialization
- Automated and Manual Serialization
- General Rules for Exchanging Serialized Data
- Using the IES mltypemap Utility to Automate Type Mapping and Serialization

## See Also

- UVM-ML TLM Communication
- UVM-ML Configuration in a Unified Hierarchy

## The Types of Data Transferred Between UVM-ML Frameworks

**Singular data types** , such as integrals, strings, and enumerated types, are passed "as is", in a raw bit format. Singular data types can be passed as separate values in configuration settings or as as class fields in TLM transactions. In either case, the main requirement for the data transfer is that the bit sizes of the values are the same on each end of the transfer.

**Serializable composite types** , such as classes and ***e*** structs, are serialized and passed by copy between frameworks according to the serialization protocol defined by the given framework adapter. These types are passed by copy because the memory layout may differ from one framework to another.The copies are maintained within each framework according to the framework's native behavior.

⚠ TLM currently supports the passing of serializable types only.

"TLM Data Types Supported by UVM-ML SystemVerilog" in UVM-ML SystemVerilog Adapter Basics Each language adapter defines the set of data types that it supports for configuration values and the set of data types that it supports for TLM transactions. These sets of data types are described in:

- "Hierarchical Configuration Data Types Supported by UVM-ML SystemVerilog" in UVM-ML SystemVerilog Adapter Basics
- "TLM Data Types Supported by UVM-ML SystemC" in UVM-ML SystemC Adapter Basics "Hierarchical Configuration Data Types Supported by UVM-SystemC" in UVM-ML SystemC Adapter Basics
- "Data Types Supported for UVM-ML e Configuration and TLM Communication" in UVM-ML e Adapter Basics

## Understanding Class Serialization and De-Serialization

Serialization and de-serialization enables you to transfer classes between languages without having to write or generate any extra code. It also enables you to include nested dynamic containers (such as dynamic arrays, TLM2 generic payload extensions, and so on) without having to create convertors.

Serialization in UVM-ML supports polymorphic subtype transactions, thus enabling an OO programming paradigm. For example, if a port interface is parameterized by a base-class transaction type, then an actual transaction can be of a derived type (sub-class).

Serialization and de-serialization in UVM-ML basically consists of:

1. Packing the fields of an object into a bit stream.
2. Passing the bit stream across a language boundary.
3. Unpacking the bits into an object on the other side.

Thus, serializable classes are passed across a language barrier in the form of a serialized stream of bits. On the consumer side, the serialized stream of bits is analyzed to recognize the object type. After the type is recognized, either a new object is allocated or an existing object is selected on the consumer side to hold the unpacked bits.

After the new object is allocated or the existing object is selected, the incoming data stream is de-serialized into that object.

The decision to allocate a new object or re-use an existing object is specific to the language adapter. Basically, it depends on whether the user-level communication protocol defines a memory management policy. Specifically, the standard TLM2 interface requires that the objects be reused because they can appear on the forward path and be sent back on the backward path. (The implications of this for UVM-SystemC are described in TLM2 Memory Management Considerations for UVM-SystemC .)

The other UVM-ML supported communication interfaces--TLM1 and configuration--pass arguments by value and thus cause allocation of a new object each time.

## Automated and Manual Serialization

Automated serialization is provided as follows:

- The predefined TLM2 class `uvm_tlm_generic_payload` is fully automated by all adapters provided with UVM-ML
- The standard UVM SystemVerilog field automation macros `` `uvm_field_* `` automatically serialize the fields of classes that extend `uvm_object`. (You must mark the fields with these macros.)
- All physical fields (%) in ***e*** are automatically serialized by default. You can override this default behavior with uvm_ml_config.tlm_pass_field() to enable other fields to be serialized as well.

You need to provide the serialization code for the following:

- All SystemC user-defined types
- SystemVerilog types derived from `uvm_tlm_generic_payload` --or other class fields (for

example, code from a UVC vendor).

Instructions for providing such serialization code is described in:

- Creating SystemVerilog Serialization Code in UVM-ML
- Creating SystemC Serialization Code in UVM-ML

## General Rules for Exchanging Serialized Data

Following are general rules for a successful exchange of serialized data:

**1) The class names on the sending and receiving ends must either match or be explicitly mapped to each other.**

The class name sent tells the receiving adapter into which type to unpack the serialized data. By default, UVM-ML assumes that that the classes match if their names match.

In SystemVerilog and ***e***, you can override this default behavior by specifying that two classes map even if the names are different. For example, if the ***e*** type "packet" maps to the SystemVerilog class "transfer", you can declare to UVM-ML that the two are the same. The constructs for doing this are described in:

- uvm_ml::set_type_match() ( SystemVerilog)
- UVM-ML e Type Mapping ( ***e*** )

If the names are not the same and are not explicitly mapped, you get a run-time error like this:

```
UVM_FATAL @ 10: reporter [MLTYPID] Bad type id in the incoming ML stream
```

**2) The size of the serialized data sent must match the size of the de-serialized data received** .

If the sizes of two mapped types differ, perhaps because of mismatched class or type definitions, you get an error message like the following:

```
[10] producer-@4: Sending packet = { data = 18 }UVM_FATAL @ 10: reporter
[UNPKSZ]
Unpacked fewer ints than were packed. This may be due to a mismatch in class
definitions across languages. Base class = packet .
```

**3) The order of fields in composite types (such as classes or structs) must be consistent.**

The field names do not need to match (or be mapped). However, it can make debugging easier if they do match.

## Using the IES mltypemap Utility to Automate Type Mapping and Serialization

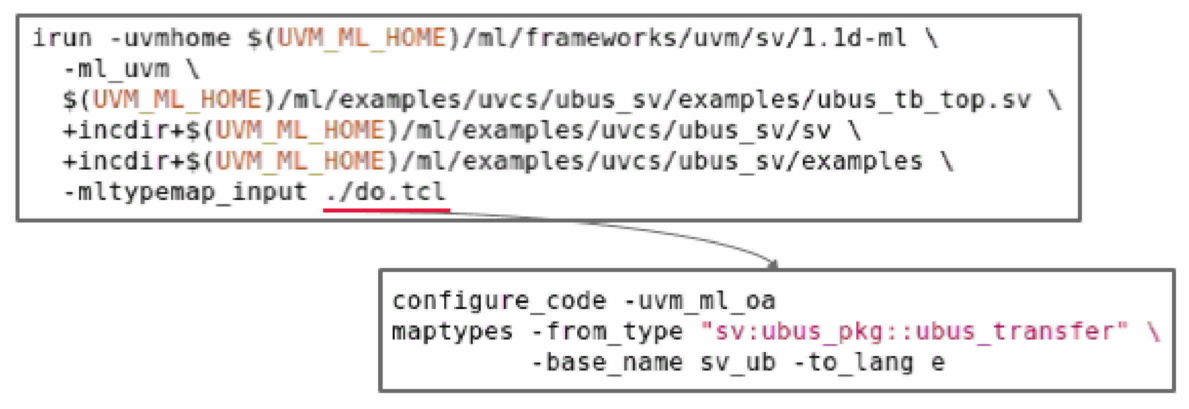
The IES automation tool **mltypemap** translates data types from one language to another. In so doing, it:

- Generates code in the target language (SystemVerilog / SystemC / ***e*** ).
- Creates matching struct/class declarations.
- Creates serialization/de-serialization code for mapped types.

### Example:

The following irun command:

- Sets UVM-ML options: `-uvmhome` and `-ml_uvm` (note that -uvmhome can also point to SystemVerilog1.2 instead of SystemVerilog1.1d).
- Compiles source type definitions.
- Executes the `mltypemap` commands in the file `do.tcl` .



```
irun -uvmhome $(UVM_ML_HOME)/ml/frameworks/uvm/sv/1.1d-ml \
  -ml_uvm \
  $(UVM_ML_HOME)/ml/examples/uvcs/ubus_sv/examples/ubus_tb_top.sv \
  +incdir+$(UVM_ML_HOME)/ml/examples/uvcs/ubus_sv/sv \
  +incdir+$(UVM_ML_HOME)/ml/examples/uvcs/ubus_sv/examples \
  -mltypemap_input ./do.tcl
```

```
configure_code -uvm_ml_oa
maptypes -from_type "sv:ubus_pkg::ubus_transfer" \
         -base_name sv_ub -to_lang e
```

### See Also

- "Type Mapping" in the *UVM-ML OA User Guide* . This section includes information about how to run mltypemap.
- The chapter "mlptypemap Utility: Automatic Generation of Data Types" in the Cadence *UVM Multi-Language Reference* .

# UVM-ML TLM Communication

UVM-ML supports communication based on the TLM1 and TLM2 interfaces. As described in Data Communication in a UVM-ML Environment , TLM transactions are passed by copy, and the copying is implemented by means of type mapping, serialization and de-serialization .

With UVM-ML, you code TLM transactions as you normally would . The calls to multi-language ports or sockets are the same as the calls used in a single-language environment. Thus, original UVC code does not need to be modified when the UVC is integrated into a multi-language environment.

There are, however, several requirements for enabling TLM communication in a multi-language environment. These requirements, plus more details about TLM communication in UVM-ML are described in the following:

- Requirements for Enabling TLM1/TLM2 Communication in a Multi-Language Environment
- More Information About the Multi-Language TLM2 Interface
- Limitations for UVM-ML TLM1/TLM2 Communication

## See Also

- Data Communication in a UVM-ML Environment
- "Multi Language Data Communication" in the *UVM-ML OA User Guide* .

## Requirements for Enabling TLM1/TLM2 Communication in a Multi-Language Environment

- **Ensure that you meet the requirements (legal data types, type mapping, and serialization) described in Data Communication in a UVM-ML Environment .**
- **Register TLM ports/sockets for multi-language communication.**

  To connect a TLM port to a port in another framework, you need to register each ports in its native framework adapter.

  UVM-ML checks the compatibility of the ports or sockets on both ends and registers the connection in its internal database. As a result of registration, the adapters create an internal channel for passing the transactions to other frameworks.

  Registration is described in the following sections:

  - SystemVerilog TLM1: uvm_ml::ml_tlm1#(T1,T2)::register() and uvm_ml::ml_tlm1#(T1,T2):register_directed()
  - SystemVerilog TLM2: uvm_ml::ml_tlm2::register()
  - SystemC TLM1: uvm_ml_register()
  - SystemC TLM2: ml_tlm2_register_initiator() and ml_tlm2_register_target() and UVM-ML SystemC Convenience Macros for Registering TLM2 Sockets

  Note that registration in ***e*** happens transparently during the normal process of defining and binding TLM ports and sockets.

- **Connect (bind) the TLM ports/sockets between frameworks.**

  Connecting the ports needs to be done in one language only and can be done in either one–or even in a different framework. For example, you could connect SystemVerilog/SystemC ports from an ***e*** test file.

  Connecting TLM ports/sockets is described in the following sections:

  - SystemVerilog: uvm_ml::connect()
  - SystemC: uvm_ml_connect()
  - e: uvm_ml.connect_names()

## More Information About the Multi-Language TLM2 Interface

The UVM-ML TLM2 interface:

- Supports multi-language communication via combined forward and backward, blocking and non-blocking transport interfaces.
- Allows passing of generic payloads and phase and timing annotation arguments through the interface methods of the sockets in different languages.
- Supports extensions of the generic payload.

### Transaction Mapping for the Backward Paths of Non-Blocking Transactions

In a single-language environment, the TLM2 transactions are passed by pointer, not by copy. Thus, a returned transaction (potentially with some modified fields) is automatically the same object as the object that was sent.

In a multi-language environment, objects cannot be passed by pointer, but rather are passed by copy, as a serialized bit stream. In this environment, ensuring that a returned transaction is the same object as the object that was sent (a process called "transaction mapping") results in runtime performance and memory costs.

At times, you might consider the cost unjustified, for example because certain sockets are used only on a forward path. In such cases, you can turn off transaction mapping. The result will be that any TLM2 transaction on the backward path is carried in different object than the object sent on the forward path.

The constructs for managing transaction mapping are described in the following sections:

- The `bit` argument for uvm_ml::connect() (SystemVerilog)
- The *map-transactions* Boolean arugment for uvm_ml_connect() (SystemC)
- any_tlm_socket.turn_transaction_mapping_off() ( ***e*** )

## Additional Considerations for Non-Blocking TLM2 Transactions

When TLM2 non-blocking transactions on the backward path are copied back to the same object that was sent on the forward path (that is, when transaction mapping is on), you need to consider the following:

- A memory manager for the transaction is required for UVM-SystemC. This requirement is described in more detail in TLM2 Memory Management Considerations for UVM-SystemC .
- If you do not execute the full phasing protocol and the transaction sent on the forward path never returns on the backward path, a memory leak will result if transaction mapping is turned on (the default).

## Limitations for UVM-ML TLM1/TLM2 Communication

UVM-ML TLM communication has the following limitations:

- Disabling (or killing) of blocking multi-language threads is not supported.
- The multi-language TLM1 interaface does not support `tlm_generic_payload` .
- The multi-language interface class type arguments (the generic payload and the phase) are passed by copy. Consequently, the changes made to argument values are not immediately visible across language boundaries. They are updated at the end of the call.

Additional language-specific limitations are described in UVM-ML SystemC Limitations Related to TLM Communication .

# UVM-ML Hierarchy

Testbenches are naturally constructed in a hierarchical manner. Single framework (single language) testbenches require hierarchical parent-child hierarchical relationships to enforce a predictable build order and testbench construction.

Multi-language testbenches, however, with "top" modules in each language, present a new challenge to handling testbench hierarchy.

Earlier multi-language solutions allowed  UVM SystemVerilog, SystemC, and ***e*** to be interconnected via TLM ports, but they did not address the issues that arise when you work in a multi-language, class-based environment .

UVM-ML provides a method for creating a single hierarchy for a multi-language environment. To do so, you instantiate the child component from the parent framework using a special UVM-ML method. This forms one, "logical" top-down hierarchy–which, in UVM-ML, is called a ***unified*** multi-language hierarchy.

## In this Chapter

This chapter provide more information about unified hierarchies. It starts first with an explanation of the benefits of single-language quasi-static hierarchies, which are used in UVM for class-based hierarchies. It then goes on to describe unified multi-language hierarchies and their benefits.

- Understanding Quasi-Static Hierarchies
- Understanding Unified Hierarchies

## Understanding Quasi-Static Hierarchies

The difference between a static and a quasi-static hierarchy are as follows:

- ***Static hierarchies*** are generated for traditional module-based testbench environments (HDL or SystemC). A static hierarchy is instantiated before the beginning of the test, during elaboration. The module-based environment is elaborated into the simulation image, and thus the design and testbench hierarchies are know at time zero.
- ***Quasi-static hierarchies*** were introduced for class-based verification environments. A quasi-static hierarchy comprises quasi-static components such as `uvm_component` in UVM-SV and UVM-SC or units in ***e***. What these components have in common is that they are created at build-time and cannot be deleted or substituted after that phase.

In contrast to dynamic classes, quasi-static components manage parent/children relationships, and each has a unique, persistent hierarchical name. Frameworks that support these qualities enable not only native-language quasi-static hierarchies but also multi-language unified quasi-static hierarchies.

### Framework Support for Quasi-Static Hierarchies

Not all the frameworks provided with UVM-ML support generation of quasi-static hierarchies.

The following frameworks ***do*** support generation of quasi-static hierarchies: UVM-SystemVerilog, UVM-e, UVM-SystemC for Incisive, and UVM-SystemC for Accelera SystemC (with addition of the patch included in the UVM-ML OA distribution).

Other vendor-specific SystemC implementations do not support generation of quasi-static hierarchies. You can still, however, generate a unified hierarchy with the portable UVM-SystemC adapter as long as the top component is in SystemC.

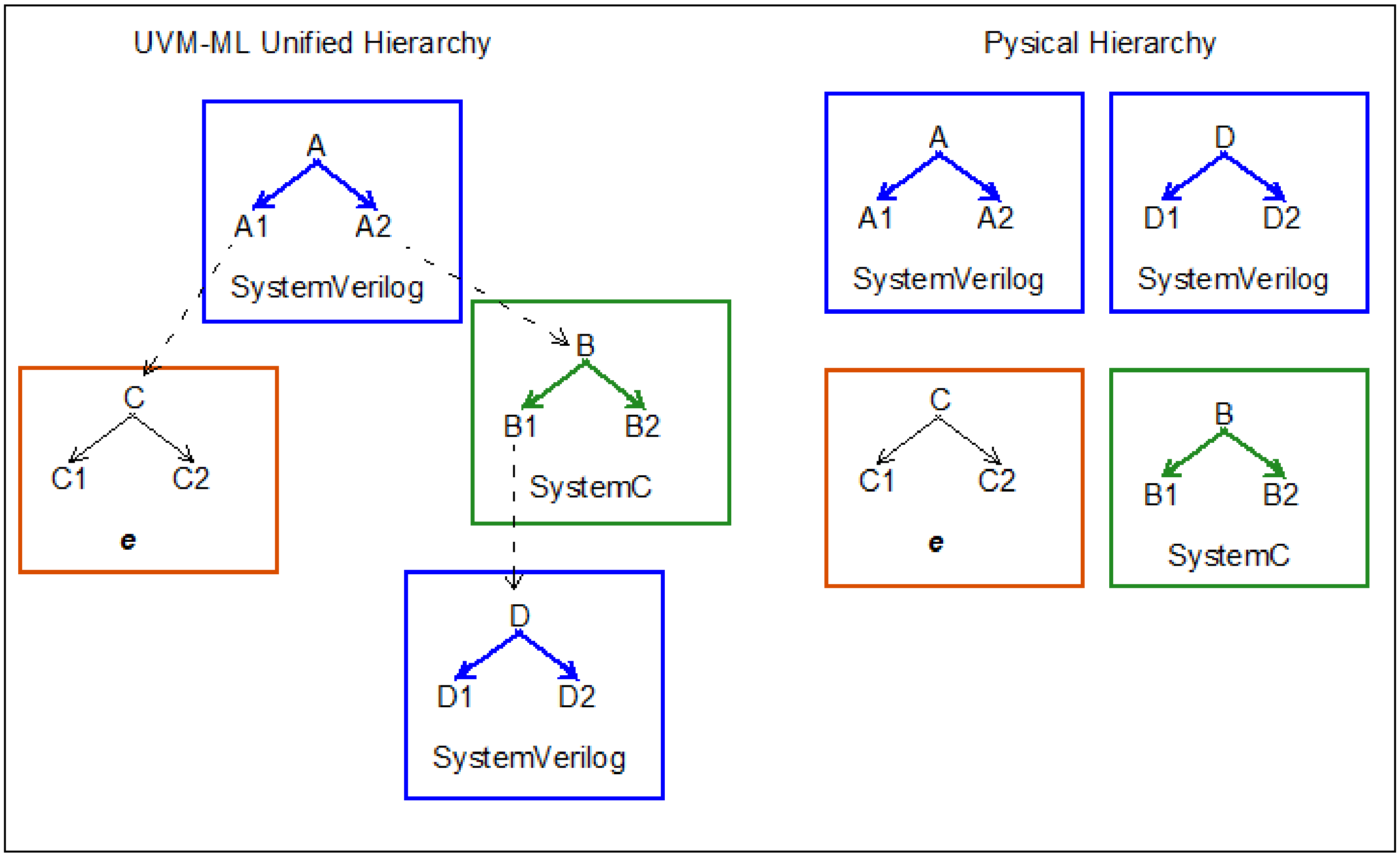
For more information about using the portable UVM-SystemC adapter and the code adaptations you might need to make to run with it, see "Running UVM-ML with Questa" in the *UVM-ML OA User Guide.*

## Understanding Unified Hierarchies

As described in UVM-ML Hierarchy , UVM-ML provides a method for creating a single, "logical" top-down hierarchy for a multi-language testbench. This method lets you instantiate a child component in one framework from a parent component in another framework, thus connecting the two. Such a hierarchy is called a **unified** multi-language hierarchy.

The following figure represents a UVM-ML unified hierarchy and its internal implementation as a physical hierarchy. In the figure:

- SystemVerilog component A1 is the hierarchical parent to the ***e*** component C.
- Likewise, SystemVerilog component A2 is the hierarchical parent to the SystemC component B, and so on.
- The dotted lines represent the connections between frameworks created by UVM-ML. The arrows in this simple figure represent hierarchical order; the arrow starts at the parent component and ends at the child component.



This figure does not represent a real-life hierarchy. Examples of real-life parent-child connections between frameworks are described in the *UVM-ML OA User Guide* in the section "Common Use

Models of Multi Language Environment" and include:

- Integrating a verification component into a testbench of a different language.
- Integrating a checker or reference model into a testbench of a different language.

Setting up a unified hierarchy requires some initial work. But the benefits in the long run are significant–after setting up a unified hierarchy, you are able to work in a multi-language environment without being concerned about the coordination and communication issues between languages that occur in a side-by-side environment.

## Unified Hierarchies Are Optional

Creating a unified hierarchy is not required for using UVM-ML. You can create an environment containing multiple tops – or a **side-by-side** hierarchy, like the physical representation in the figure above. Many UVM-ML capabilities function in side-by-side architecture, including connecting TLM ports for passing items among components.

## Creating a Unified Hierarchy

The following sections from the *UVM-ML OA User Guide* describe how to create a unified hierarchy:

- "Instantiating a SystemVerilog Component Within an e Unit"
- Instantiating an e Unit Within a SystemVerilog Component"

The language constructs used to instantiate a foreign child component are described in the following sections:

- uvm_ml_create_component() -- SystemVerilog
- child_component_proxy::create_foreign_component() --SystemVerilog
- UVM-ML SystemC Instantiation of Foreign Child Components -- SystemC
- UVM-ML e Instantiation of Foreign Child Components -- ***e***

The UVM-ML "ml/examples/features/unified_hierarchy" directory contains additional examples illustrating how to create a unified hierarchy.

## In This Section

The following subsections provide more details about unified hierarchies:

- Top-down, Hierarchical Configuration at Build-Time

- Ability to Use Relative Names to Identify Components and Ports in Another Framework
- Synchronized Phasing Enabled across Frameworks

## See Also

- Understanding Quasi-Static Hierarchies
- UVM-ML Configuration in a Unified Hierarchy

## Top-down, Hierarchical Configuration at Build-Time

Typically, build-time configuration is propagated top down--each component configures the components under it.

A unified hierarchy enables top-down configuration of the complete multi-language environment.

Such hierarchial configuration means that you can set contingent values and then override them where appropriate by the values set for a higher level instance . This is because, during th e build phase, the context automatically affects the priority property. This allows a higher ancestral parent component to override values specified by other components on lower hierarchical levels, making it possible, for example, to set default configurations in the UVC and then override them as necessary from the testbench.

For more information, see UVM-ML Configuration in a Unified Hierarchy .

## Ability to Use Relative Names to Identify Components and Ports in Another Framework

A unified UVM-ML hierarchy supports multi-language hierarchical names for child component and their ports.You form the name by concatenating the parent's path with the child's relative name--that is, “ *parent-name-in-language1.child-relative-port-name-in-language2* ”.

The use of relative names is illustrated in the code example below. As shown in the example, you do not need to keep track of a shadow SystemVerilog hierarchy in SystemC to bind the parent to its child. You need only identify the relative hierarchical port names.

In this example, a UVM-SystemVerilog UVC is instantiated in a UVM-SystemC harness layer:

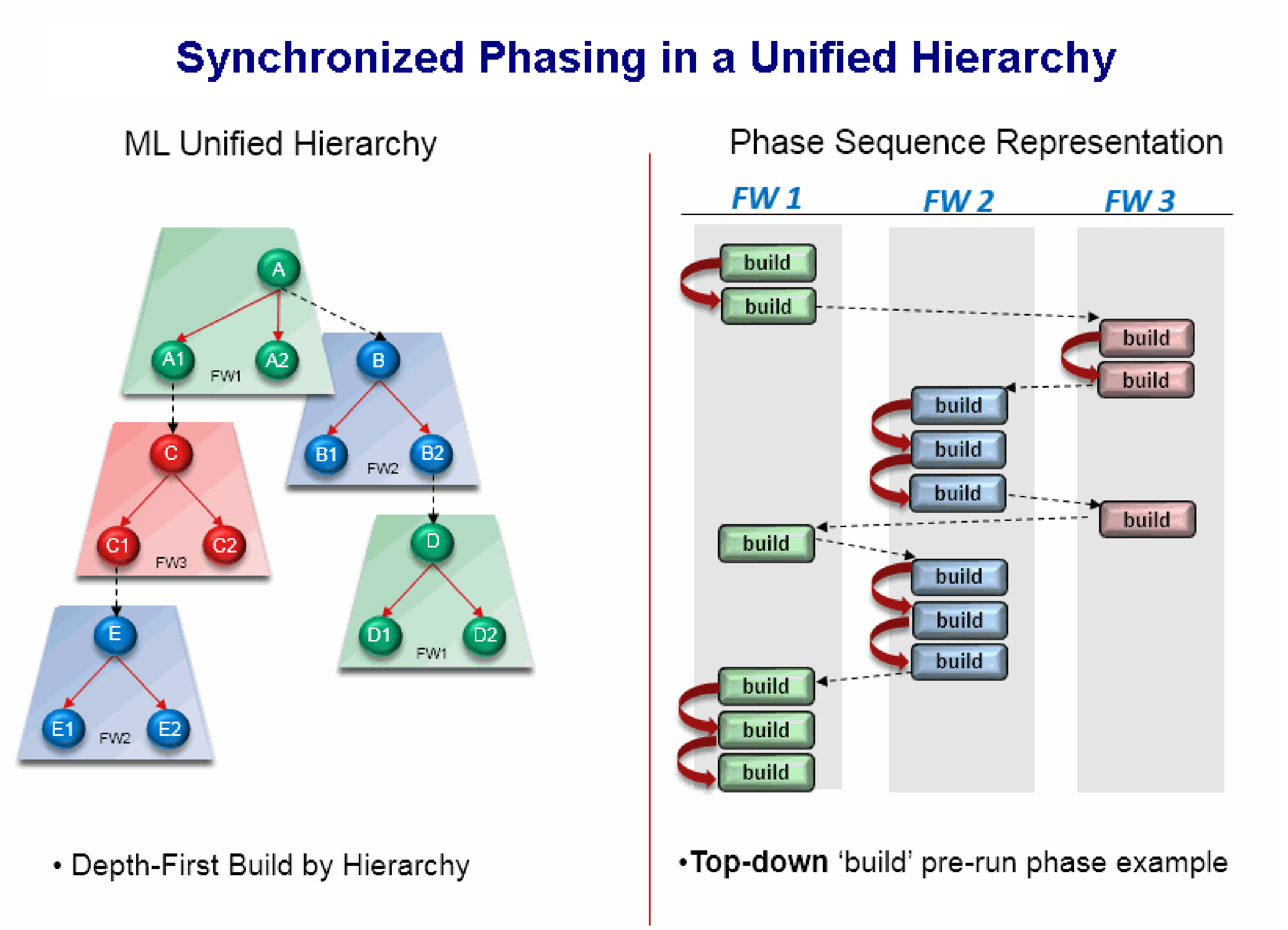
- The class member field uvc `_env` identifies the SystemVerilog child component, which is instantiated in the `sc_harness` class with the UVM-ML SystemC `uvm_ml_create_component` method. The type name of the SystemVerilog component is uvc `_env` , and the instance UVM name is uvc_ `env` .

- The SystemC TLM analysis port `aport` is a member of the `sc_harness` class. It is bound to the `control_imp` field of the corresponding SystemVerilog export with the method `uvm_ml_connect` , which takes two string arguments: the relative pathnames for the two ports:

```
class sc_harness : public uvm_component {
public:
    command_api ca;
    uvm_component * uvc_env;
    build_config_c * env_config; // Config object
    tlm_analysis_port<uvm_seq_control_base> aport;
    sc_harness(sc_module_name nm):uvm_component(nm) ,aport("aport"), uvc_env(0)
    { env_config = new build_config_c("SV","uvc_env",ACTIVE);
      uvm_ml_register(&aport);
    }
    void build_phase(uvm_phase *phase) {
        uvc_env =  uvm_ml_create_component (
                     env_config->frmw_name,
                     env_config->type_name, "uvc_env",  this);
    }
     void connect_phase(uvm_phase *phase) {
        string aexport_name = uvc _env->name()+string(".")+"control_imp";
     uvm_ml_connect ( aport.name (), aexport_name);
    }
};
```

## Synchronized Phasing Enabled across Frameworks

As described in Synchronized Phasing in a UVM-ML Environment , synchronized phasing in a unified hierarchy coordinates the phases as if the environment were in one framework. This concept is illustrated in the following figure:



## Synchronized Phasing in a Unified Hierarchy

• Depth-First Build by Hierarchy

•**Top-down** 'build' pre-run phase example

In a side-by-side architecture, phases are synchronized within individual trees–not over the entire hierarchy--with the result that you have less control over the order of phases execution.

## See Also

- Synchronized Phasing in a UVM-ML Environment

# UVM-ML Configuration in a Unified Hierarchy

Unlike single-framework or side-by-side testbenches, in which pre-run configuration values are set for one language only, UVM-ML testbenches that build a unified hierarchy can propagate pre-run configuration values from one framework tree down to all sub-trees in other frameworks.

In other words, you can configure a foreign child from its instantiator in the parent framework. In this environment, values set by the highest ancestral parent component override the values set by other components on lower hierarchical levels.

The benefits of this hierarchical configuration are that:

- You can configure UVC build-time properties (for example, number of agent instances, active or passive agent mode, and so on) natively from verification code written in the parent framework.
- You can set configuration values in the test component that will override default values set for UVCs integrated into the environment, making it easier to run tests.

To configure foreign child components, 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*** .

To retrieve a given configuration value, you also use the native UVM constructs, such as `uvm_config_db#(T)::get` in SystemVerilog or `keep uvm_config_get()` in ***e*** .

Configuration values are stored locally (in the configuration database) in each framework and are retrieved from the local cache.

The following sections describe the framework-specific constructs used to set and get configuration values in a unified hierarchy:

- UVM-ML SystemVerilog Configuration
- UVM-ML SystemC Configuration
- UVM-ML e Configuration

## Notes

- In a unified hierarchy, configuration values can be propagated to foreign child components only. You cannot, for example, configure foreign sibling components.
- Not all types can be passed to foreign child components.Types that cannot be propagated include those for which no equivalent type exists (for example, SystemVerilog virtual

interfaces) or which cannot be serialized by UVM-ML (for example, classes that do not extend `uvm_object` ). You can set such configuration values only within a given framework.

The data types that are supported for hierarchical configuration are described in the following sections:

- "Hierarchical Configuration Data Types Supported by UVM-ML SystemVerilog" in UVM-ML SystemVerilog Adapter Basics
- "Hierarchical Configuration Data Types Supported by UVM-ML SystemC" in UVM-ML SystemC Adapter Basics
- "Data Types Supported for UVM-ML e Configuration and TLM Communication" in UVM-ML e Adapter Basics

## See Also

- Unified hierarchies are described in UVM-ML Hierarchy .
- "Configuration in a Multi Language Environment" in the *UVM-ML OA User Guide* provides configuration examples.

# UVM-ML Simulation (Test) Control

When you start a simulation, you must declare the top components and/or the optional test component. UVM-ML instantiates each component you specify and constructs the tree defined by it.

This section provides more information about the start of simulation and how to manage simulation control when multiple simulators are involved:

- Tests Versus Top Entities
- Starting a UVM-ML Simulation
- Synchronizing Multiple Simulators

## Test and Top Components

When you start simulation, you can specify multiple top components and/or an optional test component for that simulation run. This section describes the difference between test and top components.

### Tests

UVM-ML adopted the concept of the test from UVM.

A test, which usually includes configuration and build information specific to the given test, can be a SystemVerilog or SystemC component or an ***e*** loadable module. Verification can involve running multiple different tests.

If the test is in SystemVerilog or SystemC, the test component receives a fixed instance name– `uvm_test_top` . This consistent naming ensures that other parts in the verification environment can access components in SystemVerilog with a full name that is fixed for all the tests associated with a specific configuration–making it easier to switch from test to test. In addition, a consistent test instance name enables merging coverage results from simulations with different tests.

If the test is in ***e*** , it is important to use a loadable module so that you do not need to recompile the code when switching from test to test.

Regardless of which language the test is in, UVM-ML begins phasing propagation from the framework to which the test component belongs.

## Tops

A top component is a SystemVerilog or SystemC component that instantiates a hierarchical tree under it.

Note that in side-by-side hierarchies, you can have multiple top components in any given framework.

It is not recommended that ***e*** modules be specified as top components. ***e*** modules are often compiled at some point to improve runtime performance, at which point, they can no longer be specified as a top component, requiring a change to code. Note that there is never a need to specify the topmost ***e*** hierarchical unit ***sys*** explicitly. It is implicitly instantiated whenever ***e*** is involved.

## See Also

- UVM-ML SystemVerilog Start of Test provides more information about test and top components and also contains multiple examples illustrating the use of top and test components.
- UVM-ML Hierarchy describes side-by-side and unified hierarchies in UVM-ML
- "Side-by-Side Multiple Tops Environment" in the *UVM-ML OA User Guide* provides additional methodology and examples for environments with multiple tops

# Starting UVM-ML Test Phases

You can start UVM-ML test phases in one of two ways:

- Procedurally, using the UVM-ML SystemVerilog task `uvm_ml_run_test()`.

  `uvm_ml_run_test()` should be used instead of the SystemVerilog `run_test()` or the `SystemC` `sc_start()`. (You cannot use `run_test()` or `sc_start()` to start a UVM-ML simulation.) You specify the top components and/or the test entity as arguments to the `uvm_ml_run_test()` task.

- With the IES `irun` command

  The IES `irun` (and `ncsim`) commands provide an alternative method to the procedural `uvm_ml_run_test()` method. When you use one of these commands, you define the top and test components using command-line options (`-uvmtest`, `-uvmtop`).

  In this case, the test phases are activated automatically, after all the global variables are initialized and before any concurrent process is executed.

### See Also

- UVM-ML SystemVerilog Start of Test
- irun -uvmtest -uvmtop

## Synchronizing Multiple Frameworks

If multiple frameworks are integrated in the simulation of a verification environment, one framework must be designated as the time master, while all other frameworks work in slave mode.

⚠ The master framework is the one that receives the simulation time from the simulator activated by the command-line.

UVM-ML allows the frameworks to be automatically synchronized on every multi-language transaction. Transactions initiated by the master framework propagate the time to the slave frameworks. It is responsibility of the slave framework adapter to adjust its time to the time of the master, and it should never advance the time beyond the master framework's time.

If, however, a slave framework needs to initiate a transaction, it must wait for the master framework to pass control to it. UVM-ML provides a wakeup request API a slave framework adapter can use to implement the wait. The ASI SystemC adapter, included in UVM-ML OA package, uses this facility if ASI SystemC version 2.3 is used. Thus, the ASI SystemC 2.3 user can apply wait statements in their code in a transparent native-language manner.

If some slave framework adapter (for example, ASI SystemC 2.2) does not automate support for wait statement, or for another reason the automated synchronization does not satisfy the user's needs, there is an explicit synchronization mechanism that can be used. From UVM-ML SystemVerilog, this can be accomplished with the special function `uvm_ml::synchronize` , which causes the SystemVerilog master framework to broadcast the current time to all the frameworks. Potentially, a slave framework can then act, in turn, as a master with respect to another slave framework.

### See Also

- UVM-ML SystemVerilog Time Synchronization of Slave Frameworks

## Synchronized Phasing in a UVM-ML Environment

Synchronized phasing is automatic for all side-by-side and unified hierarchies. You do not have to do anything to implement it.

As a first step, it is important to understand the three major groups of simulation phases:

- **Pre-run phases** initialize the system and configure, construct, and connect verification components, prior to starting environment execution.
- **Runtime phases** execute run-time tasks and include phases such as a primary run phase, reset, runtime configuration, and a runtime shutdown phase.
- **Post-run phases** extract post-run information, check the extracted data, report results, and shut down the system.

The following subsections provide more information about synchronized phasing:

- The Phasing Controller
- Controlling Synchronized Phasing in Side-by-Side Hierarchies

### See Also

- UVM-ML Hierarchy

## The Phasing Controller

UVM-ML does not itself synchronize the phases, but rather delegates phasing control to a registered phasing service provider.

If UVM-SystemVerilog is part of the environment, the UVM-ML SystemVerilog adapter automatically operates as the phasing controller and activates the phases according to the UVM-SV internal schedule.

If the integrated environment does not contain UVM-SystemVerilog, a default, internal service provider controls the phasing. This internal service provider supports the same phases as those supported by UVM-SystemVerilog.

Following is the list of supported phases.

- build
- connect
- end_of_elaboration

- start_of_simulation
- run
- reset
- configure
- main
- shutdown
- extract
- check
- report
- final

⚠ The `reset`, `configure`, `main`, and `shutdown` phases are supported in SystemVerilog and SystemC, but not in ***e***. All other phases are supported for all three languages.

## See Also

- "uvm_ml phase Tcl Command" in Incisive Debugging Commands for Use in a UVM-ML Environment

## Controlling Synchronized Phasing in Side-by-Side Hierarchies

Synchronized phasing is automatic for all side-by-side and unified hierarchies. You do not have to do anything to implement it.

For side-by-side hierarchies, however, you can influence the order in which the frameworks are called during a given phase. To do so, see UVM-ML SystemVerilog Start of Test and the irun -uvmtest -uvmtop options:

- If a "uvmtest" is identified, its framework is first in the phasing order
- Other hierarchical trees are phased in the order that their tops appear in the 'tops' array of `uvm_ml_run_test` arguments or on the command line.

# Sharing UVM Events and Barriers

With UVM-ML, you can synchronize and share UVM events and UVM barriers between the UVM-SystemVerilog and UVM-SystemC frameworks. The synchronization of events and barriers between the two frameworks is done using the standard factory-registered pools ( `uvm_event_pool` and `uvm_barrier_pool` ). Note that UVM-ML does not require that the native-language constructs be equivalent, although they are so for UVM-SystemVerilog and UVM-SystemC.

## Synchronizing UVM Events

A `uvm_event` is a wrapper class around an event, defined in the native language. The recommended methodology to allocate a `uvm_event` is to obtain it from an event pool as shown in the following SystemVerilog example:

```
class agent_a extends uvm_agent;
  uvm_event_pool e_pool;
   uvm_event control_event;
   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      e_pool = uvm_event_pool::get_pool({get_full_name(), ".events"});
   endfunction
   task run_phase(uvm_phase phase);
      ...
      control_event = e_pool.get("control_event");
      control_event.trigger();
   endtask
endclass
```

When an event pool is factory created in one framework, UVM-ML allows the pool to be associated with an equivalently named event pool in other participating frameworks. To do so, the frameworks must support `uvm_event` and `uvm_event_pool` (currently, the UVM-SystemVerilog and UVM-SystemC frameworks).

Once you get an event with the same name from a pool with the same name in another framework, the events in two frameworks are linked.

Thus, the SystemC code for synchronizing with the SystemVerilog `control_event` defined above would look like:

```
#include "uvm_ml.h"
```

```
using namespace uvm;
class top : public sc_module {
  uvm_event * control_event;
   uvm_event_pool * epool;
   ...
};
static void control_event_callback_f(uvm_event *e) {
   ...
}
public :top::top(sc_module_name name_) : sc_module(name_) {
   ...
   epool = uvm_event_pool::get_pool("uvm_test_top.env.agent_a.events");
   control_event = epool.get("control_event");
   control_event->attach(control_event_callback_f);
}
```

The order of creation between frameworks for pools or events does not matter. Upon getting a pool from the factory, the first request will create the pool if it does not exist and propagate the request to all frameworks. Subsequent requests will simply retrieve an already existing pool in its local framework, regardless of which framework created the pool first. It should also be noted that in UVM-ML, each framework keeps its own local copy of the event pool, which is mirrored to other frameworks.

Similarly, any event that is retrieved from the factory event pool will be created if it is the first request for that event and then registered in the remote framework's pool. Subsequent requests for the same event will retrieve the already created event from its local pool.

Note that this is the normal flow for event creation from event pools for single-language or multi-language environments.

## Event Pool Names and Event Names

The name of the pool can be anything as long as it is globally unique. Often a hierarchically scoped UVM name is used to guarantee uniqueness. Usually, the UVM name is provided from the component that creates the pool.

### Handling Shared Events

You can assign callbacks or wait on events in either framework and notify or trigger events as necessary. The same native-language UVM code works equally well in both single- and multi-language scenarios. The events and any associated callbacks are notified in each framework, regardless of which framework originates the notification.

The standard uvm_event `trigger()` method accepts the data argument:

```
virtual function void trigger (uvm_object data = null)
```

The function `get_trigger_data()` retrieves the data, if any, provided by the last call to trigger.

UVM-ML supports these capabilities when the data is a user-defined class extended (or derived) from the base class `uvm_object` . Data is serialized and de-serialized as is standard for UVM-ML.

## Synchronizing UVM Barriers

UVM barriers allow blocking of parallel processes or threads until a desired number of them get to the synchronization point. With UVM-ML, you can synchronize UVM barriers between frameworks similarly to how you synchronize UVM events between frameworks. The barriers ( `uvm_barrier` ) from factory created barrier pools ( `uvm_barrier_pool` ) can be shared between frameworks.

Thus, the basic steps for sharing UVM barriers between frameworks are the same as the steps described in the previous section:

1. Using the native language UVM constructs, you create or get from the factory a named `uvm_barrier_pool` in each framework where barriers are to be shared. The name of the pool can be anything as long as it is globally unique. A hierarchical UVM name of the enclosing component can be used to create the name.
2. You can then create or get `uvm_barrier` objects with the same name from the pool in each framework. The barriers in each framework are now linked to each other through the shared pool.
3. You can then apply the `uvm_barrier'` s in either framework as you normally would for a single-language environment, where control of the barrier can be initiated from any framework.

### Example

UVM-SystemVerilog:

```
uvm_barrier_pool bpool;
uvm_barrier b;
...
// Factory-create a uvm_barrier_pool with a shared unique name
bpool = uvm_barrier_pool::get_pool({get_full_name(),".bpool");
// Create (or get the previously created) named uvm_barrier object
// from the pool. Using the same name, it can be shared between the
// frameworks
b = bpool.get("mybarrier");
...
// manipulate with the barrier:
b.cancel(); // decrement waiter count
b.set_threshold(0); // clear barrier to zero
...
b.wait_for(); // wait until threshold reached
b.reset(); // reset barrier
...
```

UVM-SystemC:

```
uvm_barrier_pool * bpool;
uvm_barrier * b;
...
// Factory-create a uvm_barrier_pool with a shared agreed-upon
// unique name
bpool = uvm_barrier_pool::get_pool("top.env.bpool")
// Create (or get the previously created) named uvm_barrier object
// from the pool. Using the same name, it can be shared between the
// frameworks
bpool->get("mybarrier"); // get 'shared' barrier
...
// manipulate with the barrier:
b->raise_barrier(); // raise barrier etc.
...
b->wait(); // wait for barrier to reach threshold
```

2

# UVM-ML SystemVerilog Interface

This section describes the following:

- UVM-ML SystemVerilog Adapter Basics
- UVM-ML SystemVerilog Configuration
- UVM-ML SystemVerilog Instantiation of Foreign Child Components
- UVM-ML SystemVerilog Serialization and Type Mapping
- UVM-ML SystemVerilog TLM Interface
- UVM-ML SystemVerilog Time Synchronization of Slave Frameworks
- UVM-ML SystemVerilog Start of Test

⚠ This section primarily describes the UVM-ML-specific language constructs provided by the UVM-ML SystemVerilog adapter. It additionally contains a few descriptions of how native UVM-ML language constructs are used within a UVM-ML environment, where such descriptions are necessary to fully understand such usage.

In general, the use of native UVM-SystemVerilog language constructs is not affected by UVM-ML and requires no additional understanding on your part.

## UVM-ML SystemVerilog Adapter Basics

This section describes the following:

- Importing the UVM-ML SystemVerilog Framework Library and Adapter
- UVM-ML Framework Identifiers
- TLM Data Types Supported by UVM-ML SystemVerilog
- Hierarchical Configuration Data Types Supported by UVM-ML SystemVerilog

## Importing the UVM-ML SystemVerilog Framework Library and Adapter

UVM-ML supports the standard UVM-SystemVerilog functionality. However, several multi-language enablers (for use with UVM-ML) have been added to the standard library. Thus, the interface described in this chapter is available only if you use the UVM-SV library package that is released with UVM-ML.

In addition, you must instantiate the SystemVerilog adapter released with the UVM-ML solution to be able to use the UVM-ML SystemVerilog interface.

**To import the UVM-ML UVM-SV framework library and adapter:**

- Add the following lines to your SystemVerilog code, typically in the top-level component:

```
import uvm_pkg::*; // import the UVM-SV library package
`include "uvm_macros.svh" // macros that are part of the UVM-SV library
import uvm_ml::*; // import the UVM-ML SV adapter (this line must appear after
the above two lines)
```

## UVM-ML Framework Identifiers

Framework identifiers are used to identify the framework, for example, when you instantiate a child component that belongs to another framework.

The framework identifiers for the frameworks released with UVM-ML are as follows:

- "sv" or "uvmsv" for the UVM-SV adapter
- "sc" or "uvmsc" for any UVM-SC adapter
- "e" for the UVM- ***e*** adapter

⚠ All framework identifiers are case-insensitive.

## TLM Data Types Supported by UVM-ML SystemVerilog

UVM-ML SystemVerilog TLM transactions support classes that extend the uvm_object format. This includes the predefined classes `uvm_tlm_generic_payload` and `uvm_tlm_extension_base`.

The class properties (class data members) of the classes transmitted in TLM transactions must be of the following types:

- Integrals (signed and unsigned): any integral value less than or equal to 4096 bits (as supported by the standard `uvm_packer` ). Supported integral types are:
  - 2-state integer data types:
    - shortint
    - int
    - longint
    - byte
    - bit
  - Packed array of the 2-state integer types
  - Packed structure of the 2-state integer types
  - Packed union of the 2-state integer types
  - Enum variable
  - Time variable
- Strings
- Classes: any sub-class of `uvm_object`
- One-dimensional array (static or dynamic) or queue of one of the following:
  - The above integrals
  - `uvm_object` sub-classes
  - Strings

### See Also

- Data Communication in a UVM-ML Environment
- Creating SystemVerilog Serialization Code in UVM-ML

## Hierarchical Configuration Data Types Supported by UVM-ML SystemVerilog

All framework adapters support passing in configuration the same classes that can be passed in TLM.

In addition: any predefined singular SystemVerilog type (any data type except an unpacked structure, unpacked union, or unpacked array), including string and `uvm_bitstream_t` .

**See Also**

- Data Communication in a UVM-ML Environment
- UVM-ML SystemVerilog Configuration

# UVM-ML SystemVerilog Configuration

This section describes how to set configuration values to be propagated down through a unified hierarchy. For all other hierarchies (within one framework), set configuration values as you would for any UVM-based design.

### In This Topic

- Setting and Retrieving Configuration Values in a UVM-ML Unified Hierarchy
- The 'UVM_ML_CONFIG_DB_IMP Macro

### See Also

- UVM-ML Configuration in a Unified Hierarchy

## Setting and Retrieving Configuration Values in a UVM-ML Unified Hierarchy

Use the standard UVM `uvm_config_db#(T)::set` function or its convenience forms `uvm_config_int::set`, `uvm_config_object::set`, and `uvm_config_string::set` to set configuration values for child components belonging to other frameworks. In a unified hierarchy, these functions propagate the configuration values to the foreign components.

Similarly, use the standard UVM `uvm_config_db#(T)::get` function or its convenience forms `uvm_config_int::get`, `uvm_config_object::get`, and `uvm_config_string::get` to retrieve configuration values for child components that are set by a parent component belonging to another framework. In a unified hierarchy, these functions retrieve the configuration values set by foreign parent components.

`uvm_config_int::set()` normalizes all integral values as `uvm_bitstream_t` (as is normal for UVM).

For information about the data types that are supported by the UVM-ML SystemVerilog adapter and thus can be used to configure foreign child components, see UVM-ML SystemVerilog Adapter Basics .

## Notes

- As is normal for uvm_config_db and appropriate for a unified hierarchy, you use a relative name to identify the foreign child component.
- In a unified hierarchy, setting configuration values is supported only for foreign child components. It is not possible, for example, to configure a foreign sibling component.
- It is possible also to set a configuration value using `uvm_config_db` parameterized by a user-defined integral type or a subclass of `uvm_object` . In this case, however, you must explicitly register the user-defined type using The 'UVM_ML_CONFIG_DB_IMP Macro before setting the value in order to cause it to be propagated to other frameworks.
- If you set a configuration value for a foreign child component of a type that is not supported by UVM-ML (for example a SystemVerilog virtual interface), the value is propagated to the local UVM-SystemVerilog configuration database, but it is not propagated to the foreign child.
- `uvm_config_*::get` must be called for the configuration to happen. It is automatically called for all fields specified with `uvm_field_*` macros when `UVM_ALL_ON` is set.

## Special Consideration for Configuring e Units

When configuring an ***e*** unit that is instantiated under a when subtype, you must enter its relative instance pathname exactly as it appears in ***e*** .

For example, assume the following e code:

```
extend ACTIVE agent {
    driver: my_sequence_driver is instance;
};
extend my_env_u {
    agent: ACTIVE agent is instance;
};
```

Because “driver” is instantiated under a when subtype, its relative name appended to the ***e*** path is `ACTIVE'driver` (rather than just `driver` ). Thus, to configure it, you can use a setting like this:

```
    uvm_config_string::set(this, "*.ACTIVE'driver", "default_seq", "TEST_ML");
```

If the “ACTIVE'” prefix is omitted, the setting will not have effect.

## uvm_config_db#(uvm_object)::set() Example:

```
    class my_config_bundle_class extends uvm_object;
        int master_is_active;
        int unsigned num_of_monitors;
        `uvm_object_utils(my_config_bundle_class)
            `uvm_field_int (master_is_active, UVM_PACK | UVM_UNPACK)
            `uvm_field_int (num_of_monitors, UVM_PACK | UVM_UNPACK)
        `uvm_object_utils_end
        …
    endclass
    class env1 extends uvm_env;
        function void build_phase(uvm_phase phase);
            my_config_bundle config_object;
            super.build_phase(phase);
            config_object = new;
            config_object.master_is_active == TRUE;
            config_object.num_of_monitors ==1;
            uvm_config_db#(uvm_object)::set(this, "*my_vip_env". "config_object",
    config_object);
        endfunction
    endclass
```

## See Also

- UVM-ML Configuration in a Unified Hierarchy
- "Hierarchical Configuration Data Types Supported by UVM-ML" in UVM-ML SystemVerilog Adapter Basics
- "Configuration in a Multi-Language Environment" in the *UVM-ML OA User Guide*

# The 'UVM_ML_CONFIG_DB_IMP Macro

## Purpose

Register a user-defined type for use in configuring a foreign child component in a unified hierarchy.

## Category

UVM-ML SystemVerilog macro

## Syntax

**`UVM_ML_CONFIG_DB_IMP(** *user-defined-type* **)**

## Argument

| Argument | Description |
|---|---|
| *user-defined-type* | The user-defined integral type or `uvm_object` subclass. |

## Description

To propagate a configuration value of a user-defined type to another framework, you must register the type with the 'UVM_ML_CONFIG_DB_IMP macro before setting the value. The user-defined type must be an integral type or a subclass of `uvm_object` .

After registering the type, you can use it with `uvm_config_db#(T)::set` or `uvm_config_*::set` to set configuration values for child components belonging to other frameworks.

The language allows you to write, for example: `uvm_config_db#(reg [31:0])` or `uvm_config_db#(my_packet)` . You may want such parameter types to be specific (rather than using a generic #(int) or #(uvm_object)). This is a matter of your choice.

⚠ The macro `` `UVM_ML_CONFIG_DB_IMP `` is defined in the UVM-ML SystemVerilog adapter file `uvm_ml_macros.svh`. To use it, you must include `uvm_ml_macros.svh` explicitly.

## User-Defined Types Registration and Usage Example

```
import uvm_pkg::*; // import the UVM-SV library package that has been modified for UVM-
ML
`include "uvm_macros.svh" // macros that are part of the UVM-SV library
import uvm_ml::*; // import the UVM-ML SV adapter (this line must appear after the
above two lines)
`include "uvm_ml_macros.svh // macros defined in the UVM-ML SV adapter
....
class env2 extends uvm_env;
    function void build_phase(uvm_phase phase);
        my_config_bundle_class config_object;
        super.build_phase(phase);
        config_object = new; …
        uvm_condif_db#(my_config_bundle_class)::set(this, "*my_uvc_env".
"config_object", config_object);
        uvm_config_db#(reg[60:0])::set(this, "*producer","address",'h10000);
    endfunction
endclass
class test extends uvm_env;
    env2 sv_env;
    function void build_phase(uvm_phase phase);
        `UVM_ML_CONFIG_DB_IMP(reg [60:0]);
        `UVM_ML_CONFIG_DB_IMP(my_config_bundle_class);
        sv_env = new("sv_env", this);
    endfunction
endclass
```

## See Also

- uvm_config_db#(T)::set and Its Convenience Types uvm_config_*::set()

# UVM-ML SystemVerilog Instantiation of Foreign Child Components

This section describes the following:

- uvm_ml_create_component()
- child_component_proxy::create_foreign_component()

## uvm_ml_create_component()

### Purpose

Instantiate a child component that is implemented in a different framework (a different language ).

### Definition

functio n uvm_ml::child_component_proxy **uvm_ml_create_component(**
string *target_frmw_indicator* **,**
string *component_type_name* **,**
string *instance_name* **,**
uvm_component *parent* = null **);**

### Syntax Example

```
uvm_component uvc_top;
...
uvc_top = uvm_ml_create_component("e", "env", "uvc_top", this);
```

### Arguments

| **Argument** | **Description** |
|---|---|
| *target_frmw_indicator* | Identifies the framework that implements the foreign child component. For example, you would use:<br>● "sc" when instantiating a UVM-SystemC component.<br>● "e" when instantiating an ***e*** component. |
| *component_type_name* | The type of the instance of the foreign child component, as defined in the component's framework. |
| *instance_name* | The name of the instance of the foreign child component. |
| *parent* | The handle of the parent instance in the current framework. Use **this** to indicate the current component.<br>The default is null, which means that the parent is the topmost component `uvm_root` . |

## Description

Instantiates a child component in a different framework from the current one when you are creating a unified hierarchy .

This function returns an object of type `uvm_ml::child_component_proxy` , which is defined in the UVM-ML SystemVerilog adapter package. The class `child_component_proxy` extends the base class `uvm_component` .

The actual return value type can be defined in the user code using the base class `uvm_component` . For example:

```
uvm_component my_uvc_env;
...
my_uvc_env = uvm_ml_create_component(....);
```

Declaring the return component using the virtual base class `uvm_component` can help with future reuse, re-factoring, or refinement of the implementation. In such cases, the actual component might be once implemented as a foreign component in one configuration and as a native SystemVerilog component in another configuration. Using a common base class allows for all configurations (with no changes required).

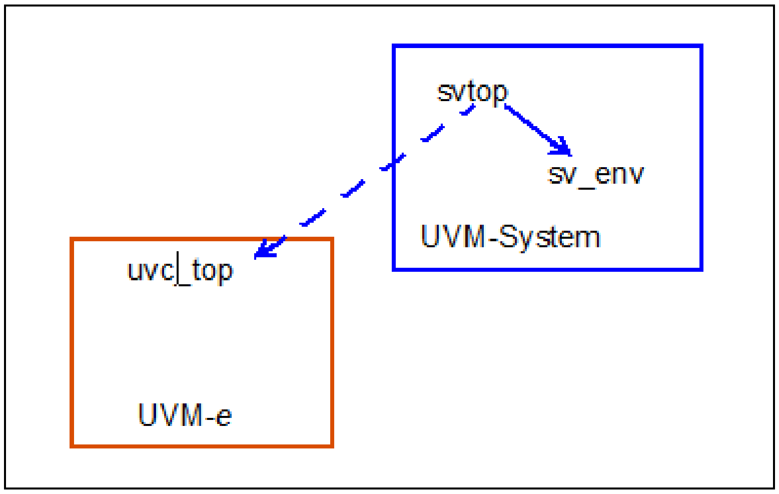
## Notes

- Instantiating a child component in a different framework is only possible if the corresponding foreign framework supports quasi-static components. For example, if your environment uses the portable UVM-SystemC adapter, then the SystemC part can only be a topmost part of the hierarchy (because the portable adapter does not generate quasi-static hierarchies). For more information, see Understanding Quasi-Static Hierarchies .
- The instantiation by a foreign language takes place in simulation time, so in cases where SystemVerilog is instantiating an ***e*** unit, features that require stub code will not work without special handling. Such features include method ports, Verilog wires, Verilog/VHDL part select, and other interface constructs. For how to handle this situation, see Specman Stub Generation for Sub-Trees in a Unified Hierarchy .

## uvm_ml_create_component() Example

The following example is taken from the `svtop.sv` file in the ml/examples/features/unified_hierarchy directory in the UVM-ML installation.

In this example, the top SystemVerilog component is `svtop` . The code in this example instantiates two child components: `sv_env` (a SystemVerilog environment component) and `uvc_top` (the top unit for an ***e*** UVC). The following figure illustrates this hierarchy:



Note that the example below assumes that user has defined and loaded some ***e*** code that defines the type `env` .

```
class svtop extends uvm_test;
    uvm_component uvc_top;
```

```
sv_env local_env;
....
function void build_phase(uvm_phase phase);
super.build_phase(phase);
....
    // create local SV env
    local_env = sv_env::type_id::create("sv_env", this);
    // create foreign ML junction node in e
    uvc_top = uvm_ml_create_component("e", "env", "uvc_top", this);
endfunction // void
....
```

## When to Use child_component_proxy::create_foreign_component()

If, instead of using the predefined base class `uvm_ml::child_component_proxy` , you need a child component proxy of a user-defined type (for example, if you are using sequence layering), then you should use child_component_proxy::create_foreign_component() instead .

## See Also

- UVM-ML Hierarchy

# child_component_proxy::create_foreign_component( )

## Purpose

Instantiate a foreign child component after the SystemVerilog proxy object of a user-defined class has been explicitly created.

## Definition

function uvm_ml::child_component_proxy:: **create_foreign_component(**
string *target_frmw_indicator* = "" **,**
string *component_type_name* = "" **);**

## Syntax Example

```
xbus_proxy_t xbus_vip = xbus_proxy_t::type_id::create("xbus_vip", this);
assert(xbus_vip.create_foreign_component(“e”, “xbus_env_u”)==1);
```

## Arguments

| Argument | Description |
|---|---|
| *target_frmw_indicator* | Identifies the framework that implements the foreign child component. For example, you would use:<br>○ "sc" when instantiating a UVM-SystemC component.<br>○ "e" when instantiating an ***e*** component. |
| *component_type_name* | The type of the instance of the foreign child component, as defined in the component's framework. |

## Description

This class member function provides an alternative to `uvm_ml_create_component` for creating a foreign child component.

`uvm_ml_create_component` is simpler because works in one step: It allocates a proxy component of the predefined type `child_component_proxy` and allocates the foreign component of the argument type.

However, some use cases require a child component proxy of a user-defined type (for example, if you are using sequence layering and need to extend the proxy). In this case, you need to allocate the user-defined sub-class component for the proxy and after that, allocate the actual foreign child component using `create_foreign_component()` .

The basic steps for accomplishing this are as follows:

1. Define a new class by extending `uvm_ml:child_component_proxy` .
2. Add the behavior specific to the foreign child component to the new class --for example, instantiate in it ports connected to the child ports, a multi language sequencer proxy, and so on.
3. Create an instance of the new class and call its `create_foreign_component()` function to instantiate the foreign child.

## create_foreign_component() Example

The following code defines a "proxy" SystemVerilog class of type `xbus_proxy_t` :

```
class xbus_proxy_t extends uvm_ml::child_component_proxy;
    ml_sequencer_proxy seqr_proxy_0; // Add proxy sequencer to interface
     function new (string name, uvm_component parent=null);
          super.new(name,parent);
     endfunction
     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
```

The following code uses this new type to instantiate the "proxy" xbus_vip and use its create_foreign_component function to instantiate the foreign child:

```
xbus_proxy_t xbus_vip = xbus_proxy_t::type_id::create("xbus_vip", this);
assert(xbus_vip.create_foreign_component("e", "xbus_env_u")==1);
```

### See Also

- uvm_ml_create_component()
- "Instantiating an e Unit Within a SystemVerilog Component" in the *UVM-ML OA User Guide* includes a complete example describing how to use `create_foreign_component()` .
- UVM-ML Hierarchy provides information about unified hierarchies.

# UVM-ML SystemVerilog Serialization and Type Mapping

This section contains the following:

- Creating SystemVerilog Serialization Code in UVM-ML
- uvm_ml::register_class_serializer()
- uvm_ml::set_type_match()

# Creating SystemVerilog Serialization Code in UVM-ML

The UVM-ML SystemVerilog adapter supports several levels of serialization automation:

- Full automation for the predefined TLM2 class uvm_tlm_generic_payload (no coding required on your part).
- Automated serialization for the fields of classes that extend `uvm_object`. You must mark the fields with the UVM-SystemVerilog `'uvm_field_*` macros. Once you do so, the serialization is automatic.

  This kind of serialization is called **in-class** .
- Serialization for SystemVerilog subclasses of `uvm_object` or `uvm_tlm_generic_payload` that you are unable to modify (for example, code from a UVC vendor). You must create special UVM-ML serializer classes for these types, either manually or with an automation tool such as Incisive mltypemp.

  This kind of serialization is called **out-of-class** because it implements the serialization and de-serialization methods without changing the original class definition.

This section gives examples of in-class and out-of-class serialization code:

- Creating SystemVerilog In-Class Serialization Code for Class Fields
- Creating SystemVerilog Out-of-Class Serialization Code

## Creating SystemVerilog In-Class Serialization Code for Class Fields

You use the standard UVM SystemVerilog field automation macros `` `uvm_field_* `` to create the in-class serialization code required to pass data fields. Among other functions, these macros implement the packing/unpacking logic for SytemVerilog data objects.

The following example uses the `` `uvm_field_int() `` macro:

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

For more information, see "Utility and Field Macros for Components and Objects" in the *Accelera UVM 1.1 Class Reference* .

## Creating SystemVerilog Out-of-Class Serialization Code

This section uses the particular case of extending the predefined base class `uvm_tlm_generic_payload` to describe how to create an out-of-class serializer for UVM-ML SystemVerilog.

⚠ The Cadence mltypemap utility automatically generates the serializer classes required for SystemVerilog TLM2 transactions. This section describes how to write the serializer code if you do not use mltypemap.

As an example, the process of creating an out-of-class serializer for an extended subclass of `uvm_tlm_generic_payload` consists of the following steps:

1. Extend the base type `uvm_ml::tlm_generic_payload_serializer` to create the new serializer class.
2. Use the predefined `serialize()` and `deserialize()` virtual methods of the predefined class `tlm_generic_payload_serializer` to serialize and deserialize the new fields added in the subclass.
3. R egister the serializer using `uvm_ml:: register_class_serializer()`.

In the following example, a serializer subclass is created for a new class `trans` derived from `uvm_tlm_generic_payload`. The `trans` class contains a user-defined integer field of `f0`. (The rest of the fields are inherited from the base class `uvm_tlm_generic_payload`.)

`uvm_ml::tlm_generic_payload_serializer` extends the UVM standard class `uvm_packer` and overrides its virtual functions. As shown in the example, you need to use `uvm_packer`'s virtual functions `pack_field_int()`, `pack_object()`, and so on in the body of the `serialize()` function.

```
// Generic payload with extension

class trans extends uvm_tlm_generic_payload;
    int unsigned f0; // adding a new field
    function string convert2string();
        return super.convert2string();
    endfunction
    `uvm_object_utils_begin(trans)
    `uvm_field_int(f0, UVM_ALL_ON);
    `uvm_object_utils_end
endclass

// Serializer for the extended generic payload
class ml_tlm2_trans_serializer extends uvm_ml::tlm_generic_payload_serializer;
    function string get_my_name();
       return "ml_tlm2_trans_serializer";
    endfunction
    // serialization method
    function void serialize(uvm_object obj);
        trans inst; // field with the extended type
         $cast(inst,obj); // cast into the proper type
         super.serialize(inst);
         pack_field_int(inst.f0, 32); // pack as an integer
    endfunction
    // deserialization method
    function void deserialize(inout uvm_object obj);
         trans inst;
         super.deserialize(obj);
         $cast(inst,obj); // cast into the proper type
         inst.f0 = unpack_field_int(32); // unpack as an integer
    endfunction
endclass
```

Note that you also need to register the serializer using `uvm_ml::register_class_serializer()` .

## See Also

- uvm_ml::register_class_serializer()

# uvm_ml::register_class_serializer()

## Purpose

Register an out-of-class serializer of type `uvm_ml::uvm_ml_class_serializer` (a subclass of `uvm_packer` ).

## Definition

function bit **register_class_serializer (**

uvm_ml::uvm_ml_class_serializer *serializer* **,**
uvm_ml::uvm_object_wrapper *sv_type* **);**

This is a uvm_ml package level task--that is., it can also be invoked as `uvm_ml::register_class_serializer(...)`.

## Syntax Example

```
function uvm_ml::uvm_ml_class_serializer get_trans_serializer();
  get_trans_serializer = trans_serializer::type_id::create();
  assert (uvm_ml::register_class_serializer(get_trans_serializer, trans::get_type()
== 1);
endfunction
```

### Arguments

| Argument | Description |
|---|---|
| *serializer* | The handle for the user-defined out-of-class serializer. |
| *sv_type* | The type ID of the subclass that needs to be serialized. |

### Description

This function creates an association between a transaction class and the corresponding serializer object. It adds the serializer object to the internal UVM-ML SV adapter registry. As the result, its functions `serialize()` or `deserialize()` are automatically invoked each time an object of the above transaction class crosses the language boundary via a TLM port or a configuration setting.

### See Also

- UVM-ML TLM Communication
- Data Communication in a UVM-ML Environment

## uvm_ml::set_type_match()

### Purpose

Map class names across framewor ks for TLM transactions.

### Definition

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

This is a uvm_ml package level task–that is, it can also be invoked as `uvm_ml::set_type_match(...)`.

## Syntax Example

```
ml_res = set_type_match( "e:main:e_packet" , "sv:my_pkg:sv_packet");
```

## Arguments

| Argument | Description |
|---|---|
| *type1, type2* | The type names for the two types to be mapped. The type names:<br>• Can be short names or fully qualified (including a namespace, package name, and so on).<br>• Must start with the framework identifier (for example, e, sv, sc).<br>• Can appear in any order.<br>Examples:<br>• "sv:my_pkg:sv_packet"<br>• "e:main:e_packet" |

## Description

This function is used to de-serialize objects. UVM-ML supports polymorphic transactions; thus, the multi-language adapter on the consumer-side needs to create a class object that matches the class of the actual argument on the producer side, at runtime. By default, the mapping between such classes assumes that the class names are the same. This function establishes correspondence between two types whose names are different..

You need to map type names only once, in one of the supported framework languages.

## Example

```
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"
);
```

### See Also

- Data Communication in a UVM-ML Environment
- "Type Mapping" in the *UVM-ML OA User Guide*

## UVM-ML SystemVerilog TLM Interface

This section describes the following:

- class ml_tlm1
- uvm_ml::ml_tlm1#(T1,T2)::register()
- uvm_ml::ml_tlm1#(T1,T2):register_directed()
- class ml_tlm2#(TRAN_T,P)
- uvm_ml::ml_tlm2::register()
- uvm_ml::connect()

### See Also

- UVM-ML TLM Communication

## class ml_tlm1

### Purpose

Class `ml_tlm1` is the container for the registration methods for TLM1 multi-language ports.

### Class Declaration

class ml_tlm1 #(type T1=uvm_object, type T2=T1);

static function void **register(** uvm_port_base #(uvm_tlm_if_base #(T1,T2)) *port* ,
string *T1_name* ="",
string *T2_name* = *T1_name* ); ...
static function void **register_directed(** uvm_port_base #(uvm_tlm_if_base #(T1,T2)) port,
ml_direction_e *direction* , string *T1_name* ="",
string *T2_name* = *T1_name* ); ...

endclass : ml_tlm1

## Methods

| | |
|---|---|
| register() | Registers TLM1 ports with the UVM-ML adapter. See uvm_ml::ml_tlm1#(T1,T2)::register() . |
| register_directed() | Registers TLM1 ports that connect hierarchically port to port, imp to imp, or export to export. See uvm_ml::ml_tlm1#(T1,T2):register_directed() . |

## uvm_ml Enumeration Type ml_direction_e

typedef enum { UNSPECIFIED_ML_DIRECTION, ML_PROVIDER, ML_PRODUCER} ml_direction_e;

UNSPECIFIED_ML_DIRECTION: Default value that indicates that no direction has been explicitly specified.

ML_PROVIDER: Indicates that the registered TLM port will be used on the transaction consuming side.

ML_PRODUCER: Indicates that the registered TLM port will be used on the transaction producing side.

## See Also

- uvm_ml::ml_tlm1#(T1,T2)::register()
- uvm_ml::ml_tlm1#(T1,T2):register_directed()

# uvm_ml::ml_tlm1#(T1,T2)::register()

## Purpose

Register UVM-ML SystemVerilog TLM1 ports to be used in connections where the data flow direction is straightforward (port-export, port-imp, export-imp).

## Definition

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

## Syntax Example

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

## Arguments

| Argument | Description |
|---|---|
| *port* | Handle of the port that is being registered. |
| *T1_name* | Optional argument. The data transaction type name. This argument can be used for checking the ports compatibility or in error messages). |
| *T2_name* | Optional argument. The data transaction type name. This argument can be used for TLM1 interfaces (for example, transport) that have two transaction type parameters (for example, request and response). |

## Description

UVM-ML multi-language TLM ports must be registered. Registering ports is the first step in connecting the port. The second step is to actually connect it with uvm_ml::connect() .

The registration must be executed at the end of the build phase, in the `phase_ended()` callback of the build. In other words, it is called before the build phase is completed, but after the foreign child components with their sub-components and ports are built. This is because you need to be able to refer to the child (as illustrated in the example).

## Example

This example illustrates how to register the port in function `phase_ended()` . This function registers a native language port.

```
class my_env extends uvm_env;
    my_agent_t agent;
    function void phase_ended(uvm_phase phase);
        if (phase.get_name() == "build") begin
            uvm_ml::ml_tlm1#(my_data_t)::register(agent.my_port);
        end
    endfunction
```

## See Also

- class ml_tlm1
- "Multi Language Data Communication" in the *UVM Multi-Language OA User Guide*
- UVM-ML TLM Communication
- "uvm_ml trace_register" in Incisive Debugging Commands for Use in a UVM-ML Environment

# uvm_ml::ml_tlm1#(T1,T2):register_directed()

## Purpose

Register UVM-ML SystemVerilog TLM1 ports when you are planning to connect it hierarchically, for example, port to port, imp to imp, or export to export. You must specify direction of the future connection explicitly, because the data flow direction cannot be automatically extracted from the port types.

## Definition

static function void **register_directed(** uvm_port_base #(uvm_tlm_if_base #(T1,T2)) *port* ,
ml_direction_e *direction* , string *T1_name* = "",
[string *T2_name* = *T1_name* ]
**);**

## Syntax Example

```
uvm_ml::ml_tlm1 #(packet,msg)::register_directed(exp1,ML_PRODUCER,"packet","msg");
```

## Arguments

| Argument | Description |
|---|---|
| *port* | Handle of the port that is being registered. |
| *direction* | One of the following:<br>● ML_PROVIDER: Indicates that the registered port will be used on the transaction consuming side.<br>● ML_PRODUCER: I ndicates that the registered port will be used on the transaction producing side. |
| *T1_name* | Optional argument. The data transaction type name. This argument can be used for checking the ports compatibility or in error messages). |
| *T2_name* | Optional argument. The data transaction type name. This argument can be used for the TLM2 interfaces (for example, transport) that have more than transaction arguments (for example, request and response).<br>The default is the *T1_name* . |

## Description

UVM-ML multi-language TLM ports must be registered. Registering ports is the first step in connecting the port. The second step is to connect it with uvm_ml::connect() .

`register_directed()` must be used if you are planning to connect hierarchically port to port, imp to imp, or export to export. In this case, you need to specify direction of the future connection explicitly, because the data flow direction cannot be automatically extracted from the port types. In this case, you need to specify the role of the port (producer or provider) to establish direction of data flow explicitly.

The registration must be executed at the end of the build phase, in the `phase_ended()` callback of the build.  In other words, it is called before the build phase is completed, but after the foreign child components with their sub-components and ports are built. This is because you need to be able to refer to the child (as illustrated in the example).

## Example

```
class consumer extends uvm_component;
  uvm_blocking_put_export #(transaction) put_export;
    ...
endclass

class top extends uvm_env;
  consumer c;
  function void phase_ended(uvm_phase phase);

    if (phase.get_name() == "build")
      uvm_ml::ml_tlm1 #(transaction)::register_directed (c.put_export, ML_PROVIDER);
  endfunction
  function void connect();
    void'(uvm_ml::connect("sctop.prod.put_port", c.put_export.get_full_name()));
  endfunction
...
endclass
```

## See Also

- class ml_tlm1
- "Multi Language Data Communication" in the *UVM-ML OA User Guide*
- UVM-ML TLM Communication
- "uvm_ml trace_register" in Incisive Debugging Commands for Use in a UVM-ML Environment

# class ml_tlm2#(TRAN_T,P)

## Purpose

Class `ml_tlm2` is the container class for the registration method for TLM2 multi-language sockets.

## Class Declaration

class ml_tlm2 #(type TRAN_T=uvm_tlm_generic_payload, type P=uvm_tlm_phase_e)
  static function void **register(** uvm_port_base #(uvm_tlm_if #(TRAN_T,P)) *socket* );
endclass : ml_tlm2

## Method

| | |
|---|---|
| register() | Registers TLM2 sockets with the UVM-ML adapter. See uvm_ml::ml_tlm2::register() . |

# uvm_ml::ml_tlm2::register()

## Purpose

Register UVM-ML SystemVerilog TLM2 sockets.

## Definition

static function void **register(** uvm_port_base#(uvm_tlm_if#(TRAN_T,P)) *socket* **);**

## Syntax Example

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

## Arguments

| Argument | Description |
|---|---|
| *socket* | Handle of the socket that is being registered. |

## Description

UVM-ML TLM2 ports must be registered. Registering ports is the first step in connecting the port. The second step is to connect it with uvm_ml::connect() .

The registration must be executed at the end of the build phase, in the `phase_ended()` callback of the build. In other words, it is called before the build phase is completed, but after the foreign child components with their sub-components and ports are built. This is because you need to be able to refer to the child (as illustrated in the example).

## Example

```
class env extends uvm_component;

  `uvm_component_utils(env)

  consumer c;

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

  function void build();
    c = new("consumer", this);
  endfunction

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

  function void connect();
    bit res;
    $display("Connecting the sockets");
```

```
      res = uvm_ml::connect("sys.producer.isocket", c.target_socket.get_full_name());
    endfunction

  endclass
```

## See Also

- class ml_tlm2#(TRAN_T,P)
- "Passing Data Items Over Language Barrier" in the *UVM-ML OA User Guide* .
- UVM-ML TLM Communication
- "uvm_ml trace_register" in Incisive Debugging Commands for Use in a UVM-ML Environment

# uvm_ml::connect()

## Purpose

Bind registered UVM-ML TLM ports or sockets located in different frameworks.

## Definition

function bit uvm_ml:: **connect(**
  string *producer_name* ,
  string *provider_name* ,
  bit *map_transactions* = 1);

This is a uvm_ml package level task--that is., it can also be invoked as `uvm_ml::connect(...)`.

## Syntax Example

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

## Arguments

| Argument | Description |
|---|---|
| *producer_name* | The full path to the port or initiator socket. |
| *provider_name* | The full path to the export/import or target socket.<br>Note: The syntax example above illustrates a recommended way to specify the path:<br>1. Set the name of the component (in this example, consumer) previously.<br>2. Concatenate the full path with port/socket names.<br>The component name can be obtained in different ways. if it is in a unified hierarchy, you can use a relative name (as is the case in the syntax example above). If it is in a side-by-side hierarchy, the component path must be absolute (for example, "sys.my_vip.Passive'agent"). |
| *map_transactions* | Turns transaction mapping on or off. The default is on.<br>For more information about transaction mapping, see "More Information About the Multi-Language TLM2 Interface" in UVM-ML TLM Communication . |

## Description

Binds ports or sockets that were registered as UVM-ML connections. You must provide the full path of the port and export/import, or initiator socket and target socket. UVM-ML checks the compatibility of the two ends and registers the connection in its internal database.

Binding the ports needs to be done in one language only and can be done in either one–or even in a different framework. For example, you could bind SystemVerilog/SystemC ports from an ***e*** test file.

This function returns 1 if the connection succeeded and 0 if fails. You can check the return value and call `uvm_error` to issue an error message with a proper source reference.

## Example

```
class env extends uvm_component;

  `uvm_component_utils(env)
  consumer c;

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

  function void build();
    c = new("consumer", this);
  endfunction

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

  function void connect();
    bit res;
    $display("Connecting the sockets");
    res = uvm_ml::connect("sys.producer.isocket", c.target_socket.get_full_name());
  endfunction

endclass
```

## See Also

- class ml_tlm2#(TRAN_T,P)
- "Passing Data Items Over Language Barrier" in the *UVM-ML OA User Guide*
- UVM-ML TLM Communication
- "uvm_ml trace_register" in Incisive Debugging Commands for Use in a UVM-ML Environment

# UVM-ML SystemVerilog Time Synchronization of Slave Frameworks

The UVM-ML SystemVerilog adapter provides the following API for time synchronization of slave frameworks.

## uvm_ml::synchronize()

### Purpose

Enable a slave framework to initiate transactions.

### Definition

task **synchronize();**

This is a uvm_ml package level task--that is., it can also be invoked as `uvm_ml::synchronize()`.

### Syntax Example

```
    uvm_ml::synchronize();
```

### Description

As described in "Synchronizing Multiple Frameworks" in UVM-ML Simulation (Test) Control , if multiple frameworks are involved in the simulation of a verification environment, one framework must be designated as the time master, while all other frameworks work in slave mode. If, however, a slave framework needs to initiate a transaction, it must wait for the master framework to pass control to the slave framework.

`synchronize()` provides an API for synchronizing slave frameworks when SystemVerilog is the master framework. This task causes SystemVerilog to broadcast the time to all the frameworks. A slave framework can then act to advance its simulator time wheel to the time point specified by the master. While advancing the time wheel, the slave engine can activate suspended processes/threads, evaluates the events, and so on.

The slave's threads can call the master back via non-blocking functions only. The blocking call chain must always be initiated by the master framework. Thus, it is an error to call a master's blocking task from the slave framework's process, activated by the explicit synchronization call.

⚠ The strobe rate should be set to the minimal needed frequency to minimize the runtime performance overhead.

## Example

```
task run_phase(uvm_phase phase);
    while(1) begin
        #1 uvm_ml::synchronize();
    end
endtask
```

## Notes

- ASI SystemC adapter is the only slave framework adapter, distributed as a part of the UVM-ML OA package. This adapter automatically synchronizes with the simulators, using the API provided by UVM-ML, if ASI SystemC version 2.3 is used..Hence, using uvm_ml::synchronize() .is not necessary when working with ASI SystemC 2.3. The UVM-ML "ml/examples/features/synchronization/sc_sv" directory contains examples that demonstrate such automated synchronization.

## Limitations

- SystemVerilog simulation frameworks currently do not support slave mode. The SystemVerilog frameworks can only act as master simulators.
- The Specman/ ***e*** framework does not support direct synchronization in any role. It requires an explicit synchronization of Specman with a simulator using an ***e*** mechanism such as `wait delay` or a port callback.
- ASI SystemC 2.2 requires usage of uvm_ml::synchronize if SystemC needs to initiate transaction after some delay.

# UVM-ML SystemVerilog Start of Test

UVM-ML SystemVerilog provides the following task for starting simulation.

## uvm_ml::uvm_ml_run_test()

### Purpose

Instantiate the multi-language top components and start a UVM-ML simulation from the UVM-ML SystemVerilog adapter.

### Definition

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

This is a uvm_ml package level task (that is, it can also be invoked as `uvm_ml::uvm_ml_run_test(...)` )

## Arguments

| Argument | Description |
|---|---|
| *tops[]* | This argument is a dynamic array of top component identifiers. (See examples below.)<br><br>Normally it should be used only for defining SystemVerilog or SystemC top components. If an ***e*** top identifier is included in this array, it represents a loadable ***e*** module. This is allowed but not recommended because an ***e*** UVC may be compiled at some point for better runtime performance, and that will require exclusion of this identifier from the *tops* array. Note also that the topmost ***e*** hierarchical unit ***sys*** is implicitly instantiated, whenever ***e*** is involved. Hence, it should not be specified explicitly.<br><br>In case you do not want to specify any tops (that means, you want the entire testbench hierarchy to be instantiated under the "test" component, as it is recommended by UVM SV), you should pass an array with one element set to be an empty string (see example #1 below).<br><br>When you define the array of top components:<br>● Identify each top component by framework ID and the type of the top component separated by a colon (:), for example `SV:svtop` .<br>● If the instance name of the top component is different from the type name, then you must append it after the type name, for example `SV:svtop:my_top` . |
| *test* | This argument represents a SystemVerilog/SystemC component or an e loadable module that is specific for a given test. Verification may involve multiple different tests. You can override the test argument via the standard UVM SV command line option +UVM_TESTNAME=[frmw-identifier:]test-name. The default framework identifier is “SV”. Example: `+UVM_TESTNAME=svtop`.<br><br>If the test is in SystemVerilog or SystemC, the test component will automatically receive a predefined instance name: *uvm_test_top* . This fixed instance name enables re-use of invariant hierarchical paths in the testbench when running different tests.<br><br>If the test is in ***e*** , it is recommended to use it as a loadable ***e*** module, so that you don't need to re-compile the code when switching from test to test. In this case, you can provide this loadable ***e*** module as the *test* argument. |

Note: An ***e*** top module can be loaded or compiled independently from the tops listed here.

## Description

This task instantiates top components (or loads an ***e*** module) and activates the UVM phases for all involved frameworks.

It is recommended to invoke this task from an initial block.

`uvm_ml_run_test()` passes the arguments and control to the UVM-ML SV adapter. The adapter serves, by default, as a phasing provider for all frameworks. It calls UVM SV and activates the regular test phases exactly like the native uvm_run_test() task. However, unlike UVM SV, it does not call the phase callbacks for all UVM components immediately. Instead, it notifies the UVM-ML implementation library about each phase start and end.

The UVM-ML implementation iterates over all top components (including *sys*, if ***e*** is involved) and invokes the framework, associated with each top. There may be only one top component (unified hierarchy) or few of them (side-by-side hierarchies). This process allows each framework to call its' component phase callbacks before switching to the next phase. Additionally, it preserves the hierarchical order (top-down or bottom-up, depending on the phase) of the callbacks in a unified hierarchy. Preservation of the order is particularly important if the testbench applies hierarchical configuration during the build phase.

## Notes

- `uvm_ml_run_test()` replaces UVM SystemVerilog `run_test(), SystemC sc_start()` and e *test* command. In other words, you cannot call `run_test(), sc_start()` or e *test* in your testbench code if you are using UVM-ML .

- Currently, there is no similar capability implemented in the UVM-ML SystemC or ***e*** adapters.

- IES supports also the following alternative to `uvm_ml_run_test()`. The simulator supports command line switches ( `-uvmtop` / `-uvmtest` ) that are equivalent to the arguments of `uvm_ml_run_test()`. When any of this arguments is specified, `uvm_ml_run_test()` should not be invoked by the user code. Instead, the simulator activates the same functionality automatically, after all the global variables are initialized and before execution of any concurrent process.

## Unified Hierarchy examples

The following example shows a test invocation for a single hierarchy tree, where the top component is the SystemVerilog test. `test7` or any other `test<num>` denote any single test out of many.

```
module topmodule;
initial begin
    string tops[1]; // to represent an empty list of tops
    tops[0] = "";   // the top will be the test7, so there is no other top
    uvm_ml_run_test(tops, "SV:test7"); // the test in SystemVerilog
end
```

**Note:** remember that if the test is in SystemVerilog, the instance name *uvm_test_top* is automatically assigned to the test component.

If the top framework and test was in ***e***, rather than SystemVerilog, only the fifth line would change

```
    uvm_ml_run_test(tops, "e:test8.e");  // the test is an e module
```

And if the top framework and test was in SystemC, the fifth line would then be like this

```
    uvm_ml_run_test(tops, "SC:test9");  // the test in SystemC
```

## Side By Side examples

If the test bench top component was in SystemC, and the SystemVerilog test was instantiated side-by-side with the test bench, the two lines before last would change to:

```
tops[0] = "SC:my_tb";   // the explicit SystemC top
uvm_ml_run_test(tops, "SV:test1");  // the test is in SystemVerilog
```

The following example creates a side-by-side hierarchy of three frameworks, with a test in ***e*** .

Note that in this example, the instance names of the top components are identical to their type names

```
module topmodule;
initial begin
    string tops[2];
    tops[0] = "SV:svtop"; // the top SystemVerilog uvm_component
    tops[1] = "SC:sctop"; // the top SystemC uvm_component or module
    uvm_ml_run_test(tops, "e:test1.e"); // a test in e
end
```

```
endmodule
```

## If You Are Using a Portable UVM-SC adapter

If you are using the UVM-ML SystemC portable adapter (for example, with Questa SystemC), the SystemC hierarchy must be constructed statically, starting from an sc_module instantiated under `sc_main`. Accordingly, all the path strings must start with “sc_main”. For example:

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

Remember also that the path to SystemC components must be adjusted according to the simulator-speciifc path notation, for example using ‘/’ instead of ‘.’ as the delimiter for Questa.

## See Also

- irun -uvmtest -uvmtop
- UVM-ML Hierarchy
- UVM-ML Simulation (Test) Control

3

# UVM-ML SystemC Interface

This section describes the following:

- UVM-ML SystemC Adapter Basics
- UVM-ML SystemC TLM Interface
- UVM-ML SystemC Serialization
- UVM-ML SystemC Instantiation of Foreign Child Components
- UVM-ML SystemC Configuration
- UVM-ML SystemC Debugging Utilities

## UVM-ML SystemC Adapter Basics

UVM-ML is currently released with three SystemC adapters:

- UVM-ML SystemC adapter for Incisive
- UVM-ML SystemC adapter for the patched ASI-SystemC
- UVM-ML Portable adapter for the standard ASI-SystemC or vendor-specific SystemC simulator

The UVM-ML SystemC adapter for Incisive and ASI-SystemC can be used either with standard SystemC or together with the UVM-SystemC library that is included with the UVM-ML release. Usage with standard SystemC is limited to support of multi-language TLM2 communication; usage with the UVM-SystemC library enables a broad scope of UVM capabilities.

This section contains the following:

- Including the UVM-ML SystemC Adapter in Your Environment
- UVM-ML Framework Identifiers
- TLM Data Types Supported by UVM-ML SystemC
- Hierarchical Configuration Data Types Supported by UVM-ML SystemC

### See Also

- "General Instructions About the Portable Adapter in "Running UVM-ML with Questa" in the *UVM-ML OA User Guide*

## Including the UVM-ML SystemC Adapter in Your Environment

To instantiate any of the UVM-ML SystemC adapters, add the following lines to the SystemC code in the top-level component:

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

If you are using multi-language TLM2, also add the following:

```
#include "ml_tlm2.h"
```

## UVM-ML Framework Identifiers

Framework identifiers are used to identify the framework, for example, when you instantiate a child component that belongs to another framework.

The framework identifiers for the frameworks released with UVM-ML are as follows:

- "sv" or "uvmsv" for the UVM-SV adapter
- "sc" or "uvmsc" for any UVM-SC adapter
- "e" for the UVM- ***e*** adapter

⚠ All framework identifiers are case-insensitive.

## TLM Data Types Supported by UVM-ML SystemC

UVM-ML SystemC TLM1 transactions support classes that are derived from `uvm_object` (defined in the UVM-SystemC library).

UVM-ML SystemC analysis interfaces support classes that are derived from `uvm_object or from the standard tlm_generic_payload (with extensions derived from tlm_extension_base).`

UVM-ML SystemC TLM2 transactions support classes that are derived from the standard `tlm_generic_payload` or `tlm_extension_base`.

For the TLM1 interfaces, the transaction class definition must contain implementation of the `do_pack()` and `do_unpack()` methods.These methods should be implemented with help of the predefined streaming operators “>>” and “<<”. The predefined streaming operators support the following types:

- `bool`
- `char` (signed and unsigned)
- `short` (signed and unsigned)
- `int` (signed and unsigned)
- `long` (signed and unsigned)
- `long long` (signed and unsigned)
- `std::string`
- `char *` (for null-terminated C strings only)
- Classes derived from `uvm_object`
- `sc_logic` (note that only logic values 0 and 1 are serialized)
- Bit vectors, which are classes derived from class `sc_bv_base`
- Logic vectors, which are classes derived from class `sc_lv_base` (note that only logic values 0 and 1 are serialized)
- Limited-precision integers, which are classes derived from class `sc_int_base` or class `sc_uint_base`
- Finite-precision integers, which are classes derived from class `sc_signed` or class `sc_unsigned`
- `std::vector<T>` , where T is any of the above types

Multi-language TLM2 supports these same types for fields in user-defined TLM extension classes (derived from tlm_extension_base).

## See Also

- Data Communication in a UVM-ML Environment
- Creating SystemC Serialization Code in UVM-ML

## Hierarchical Configuration Data Types Supported by UVM-ML SystemC

All framework adapters support passing in configuration the same classes that can be passed in TLM.

In addition:

- C string (null terminated byte array)
- Any type that can be implicitly converted to `sc_bv_base` ( `sc_int` , `sc_uint` , `sc_signed` , `sc_unsigned` , `sc_bigint` , `sc_biguint` , `int` , unsigned `int` , and so on)

### See Also

- Data Communication in a UVM-ML Environment

# UVM-ML SystemC TLM Interface

This section contains the following:

- uvm_ml_register()
- ml_tlm2_register_initiator() and ml_tlm2_register_target()
- uvm_ml_connect()
- UVM-ML SystemC Convenience Macros for Registering TLM2 Sockets
- TLM2 Memory Management Considerations for UVM-SystemC
- UVM-ML SystemC Limitations Related to TLM Communication

### See Also

- UVM-ML TLM Communication

# uvm_ml_register()

## Purpose

Register TLM1 ports.

## Definition

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

template <typename T>
void **uvm_ml_register(**
 sc_core::sc_export<tlm::tlm_analysis_if< *T* > >* p **)**

template <typename T>
void **uvm_ml_register(**
 tlm::tlm_analysis_port< *T* >* p **)**

template <typename REQ, typename RSP, int N, sc_core::sc_port_policy POL>
void **uvm_ml_register(**
 sc_core::sc_port<tlm::tlm_transport_if< *REQ* , *RSP* >, *N* , *POL* >* p **)**

template <typename REQ, typename RSP>
void **uvm_ml_register(**
 sc_core::sc_export<tlm::tlm_transport_if< *REQ* , *RSP* > >* p **)**

## Parameters

| Parameter | Description |
|---|---|
| *T* | The type of the transaction class. |
| *N* | The maximum number of targets. |
| *POL* | The policy that determines the rules for binding multiports and the rules for unbound ports. Legal values:<br>• SC_ONE_OR_MORE_BOUND<br>• SC_ZERO_OR_MORE_BOUND (the default)<br>• SC_ALL_BOUND |
| *REQ* | Request type. |
| *RSP* | Response type. |

## Description

Every TLM1 port that is to be connected in another framework must be registered as a UVM-ML port using `uvm_ml_register()` .

## Example

The following example registers a TLM1 put port in the `before_end_of_elaboration` phase (or in UVM-SC, you can use the `build` phase).

```
void before_end_of_elaboration() {
    uvm_ml::uvm_ml_register(&sc_env.prod.put_port);
}
```

Note that it is unnecessary to specify the template parameters explcitly because they are automatically extracted from the type of the port argument.

## Notes

**uvm_ml_register(** sc_port<tlm::tlm_analysis_if< *T* >, *N* , *POL* >* p **)** is currently not supported for *tlm_generic_payload* transaction types. That means, only **uvm_ml_register(** tlm::tlm_analysis_port< *T* >* p **)** is supported if T is *tlm_generic_payload* or its subtype. The implication of this limitation is that it is currently impossible to use an analysis port with a specific number of ports (N) and an explicitly specified policy (POL).

## See Also

- "uvm_ml trace_register" in Incisive Debugging Commands for Use in a UVM-ML Environment
- F or information that is specific to the portable SystemC adapter, see "Running UVM-ML with Questa" in the *UVM-ML OA User Guide* .

# ml_tlm2_register_initiator() and ml_tlm2_register_target()

## Purpose

Register TLM2 sockets

## Definition

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

t emplate <class REQ, unsigned int BUSWIDTH, typename TYPES>

std::string **ml_tlm2_register_target(** const sc_module& *containing_module* ,
tlm_target_socket<BUSWIDTH,TYPES> & *socket_instance* ,
const std::string & *target_socket_name* ,
const std::string & *cur_context_name* **)**

## Parameters

| **Parameter** | **Description** |
|---|---|
| *containing_module* | The SystemC module in which the socket is instantiated. |
| *socket_instance* | The socket instance. |
| *target_socket_name* | The socket name. |
| *cur_context_name* | The name of the SystemC module where registration is done. |

## Description

The initiator and target sockets for a TLM2 transaction must be registered to enable a multi-language connection.

The registration is templated with:

- REQ - the type of the transaction
- BUSWIDTH - the bit width of the transaction
- TYPES- the protocol types for the interface

These functions return a string value that is a full path of the socket that must be used when actually connecting the sockets by names (see uvm_ml_connect() ).

## Example

```
void before_end_of_elaboration() {
    ml_tlm2_register_target <tran_t, buswidth> (this, b_tsocket, "b_tsocket", this-
>name());
...
```

## See Also

- UVM-ML SystemC Convenience Macros for Registering TLM2 Sockets
- "uvm_ml trace_register" in Incisive Debugging Commands for Use in a UVM-ML Environment

# uvm_ml_connect()

## Purpose

Bind registered UVM-ML TLM ports or sockets located in different frameworks.

## Definition

void **uvm_ml_connect(**
  const std::string & *producer_name* ,
  const std::string & *provider_name* ,
  bool *map_transactions* = true
**);**

## Syntax Example

```
    uvm_ml::uvm_ml_connect("sys.u.my_outport" , sc_env.cons.my_export.name ()) ;
```

## Arguments

| Argument | Description |
|---|---|
| *producer_name* | The full path to the port or initiator socket. |
| *provider_name* | The full path to the export/import or target socket. |
| *map_transactions* | Turns transaction mapping on or off. The default is on (TRUE).<br>For more information about transaction mapping, see "More Information About the Multi-Language TLM2 Interface" in UVM-ML TLM Communication . |

## Description

Binds  ports or sockets that were registered as UVM-ML connections. You must provide the full path of the port and export/import, or initiator socket and target socket. UVM-ML checks the compatibility of the two ends and registers the connection in its internal database.

Binding the ports needs to be done in one language only and can be done in either one–or even in a different framework. For example, you could bind SystemVerilog/SystemC ports from an ***e*** test file.

## Example

The following example connects an e port to a SystemC export:

```
std::string full_initiator_b_socket_name = ML_TLM2_REGISTER_INITIATOR(prod,
PAYLOAD_TYPE, b_isocket , 32);
uvm_ml_connect(full_initiator_b_socket_name, "svtop.sv_env.consumer.b_target_socket");
```

## See Also

- "uvm_ml trace_register" in Incisive Debugging Commands for Use in a UVM-ML Environment

# UVM-ML SystemC Convenience Macros for Registering TLM2 Sockets

## Purpose

Register TLM2 sockets

## Definition

**target_name = ML_TLM2_REGISTER_TARGET(** *module_i* , *tran_t* , *sckt* , *buswidth* **)**
**initiator_name = ML_TLM2_REGISTER_INITIATOR(** *module_i* , *tran_t* , *sckt* , *buswidth* **)**

## Arguments

| Argument | Description |
|---|---|
| *module_i* | The component where the socket is instantiated. |
| *tran_t* | The type of the transaction. |
| *sckt* | The socket instance. |
| *buswidth* | The bit width of the socket. |

## Description

For convenience, there are macros defined for TLM2 socket registration. The TLM2 registration macros are defined in `ml_tlm2.h`.

The return value, `initiator_name` (or `target_name` returns the full path to this socket to be used later, in a connect statement.

## Example

The following example registers a TLM1 put port and a TLM2 initiator socket in the `before_end_of_elaboration` phase (or in UVM-SC, you can use the `build` phase)

```
void before_end_of_elaboration() {
    initiator_name = ML_TLM2_REGISTER_INITIATOR(i,tlm_generic_payload,isocket,32);
}
```

## TLM2 Memory Management Considerations for UVM-SystemC

You cannot control allocation of the SystemC incoming transactions (directed, for example, from ***e*** to SystemC). Thus, the UVM-ML TLM2 implementation manages memory (allocation and de-allocation) by providing an explicit memory manager per type of transaction. The transaction is returned to the memory manager’s pool at the end of the incoming call.

If the `map_transactions` argument of `uvm_ml_connect` is not explicitly set to off, he non-blocking transactions on the backward path are copied back to the same object that was sent on the forward path. This feature requires special attention:

- In SystemC, this mode requires the presence of a memory manager for the transaction (see section 14.5, "Generic Payload Memory Management" in the *IEEE Standard for Standard SystemC® Language Reference Manual 1666-2011* ).
- If you do not execute the full phasing protocol and the transactions sent on the forward path never return on the backward path, a memory leak will result.

Note that TLM 2.0 does not support a memory manager for `tlm_phase` objects. Thus, the UVM-ML TLM2 implementation reuses the objects, allocated for the incoming SystemC calls, using an internal pool. Do not use these objects, except in the context of the functions `b_transport()` and `nb_transport_fw()` .

## UVM-ML SystemC Limitations Related to TLM Communication

- The new field `m_gp_option` which was added to the `tlm_generic_payload` in the ASI-SystemC version 2.3 (see the *IEEE Standard for Standard SystemC® Language Reference Manual 1666-2011* ) is not supported at this time.
- UVM-ML hierarchical binding is not supported on the SystemC side. This means that you cannot bind a SystemC export to another SystemC export and to a SystemVerilog port.

## UVM-ML SystemC Serialization

This section contains the following:

- Creating SystemC Serialization Code in UVM-ML
- do_pack() and do_unpack()
- ML_TLM2_GP_BEGIN, ML_TLM_FIELD, and ML_TLM2_GP_END Macros

### See Also

- Data Communication in a UVM-ML Environment

# Creating SystemC Serialization Code in UVM-ML

The UVM-ML SystemC adapter supports several levels of serialization automation:

- Full automation for the predefined TLM2 class `tlm_generic_payload` (no coding required on your part).
- Partially automated serialization for SystemC subclasses of `tlm_generic_payload` or `tlm_extension_base`. You must define special UVM-ML serializer classes for these types with an automation tool such as Incisive mltypemap or by using macros like `ML_TLM2_GP_BEGIN`, `ML_TLM_FIELD`, `ML_TLM2_GP_END`, `ML_TLM2_GP_EXT_BEGIN`, `ML_TLM2_GP_EXT_END`, and so on.

  This kind of serialization is called **out-of-class** because it implements the serialization and de-serialization methods without changing the original class definition.
- Manual serialization for the fields of classes derived from uvm_object. You must provide the do_pack () and do_unpack() methods explicitly for in-class serialization for data objects, taking advantage of the streaming operator >> for unpacking or << for packing. Again, you can also use Incisive mltypemap to generate these methods for the output classes.

## See Also

- ML_TLM2_GP_BEGIN, ML_TLM_FIELD, and ML_TLM2_GP_END Macros
- do_pack() and do_unpack()

# do_pack() and do_unpack()

## Purpose

Serialize or de-serialize a UVM SystemC class.

## Definition

This is a definition of pure virtual methods in the abstract base class `uvm_object`.

```
namespace uvm {
class uvm_object : public uvm_typed {
public:
   virtual void do_pack(uvm_packer& p) const = 0;
   virtual void do_unpack(uvm_packer& p) = 0;

};
}
// namespace uvm
```

## Description

You must define the `do_pack` () and `do_unpack()` methods explicitly for in-class serialization for the actual classes, taking advantage of the streaming operator >> for unpacking and << for packing. The `uvm_packer` class is the packer object actually responsible for handling the serialized data.

The `do_pack()` and `do_unpack()` functions take an `uvm_packer&` as an argument. The `uvm_packer` has a << operator for packing and a >> operator for unpacking. The `do_pack()` / `do_unpack()` functions should list the fields to be packed/unpacked. You can string the fields together or put them on separate lines:

```
(*p) << field1 << field2;
(*p) << field3;
```

Each argument to the << and >> operators must be either an `uvm_object` or one of the supported primitive types: `bool` , `char` , `unsigned char` , `short` , `unsigned short` , `int` , `unsigned int ' long` , `unsigned long` , `long long` , `unsigned long long` , `sc_logic` , `sc_bv` , `sc_lv` , `sc_int` , `sc_uint` , `sc_bigint` , `sc_bigint` . If a field is an `uvm_object` , you can pass it as a pointer or as an object. Passing it as a pointer preserves th derived class information. However, when crossing the language boundary, the object is passed by value only.

If a field is not of one of the supported types, it must be converted to one or more objects of supported types. For example, if a field is a structure that contains subfields of supported types, you can pass the subfields as a pointer:

```
(*p) << field4.i1 << field4.i2 << field5;
```

The pack and unpack functions preserve derived type information. For example, if you declare a class derived from header, as follows:

```
class derivedheader : public header{
public:
    UVM_OBJECT_UTILS(derivedheader)
    derivedheader() { s = "dddd"; }
    ...
  };
  UVM_OBJECT_REGISTER(derivedheader)
```

and the packed class derived from uvm_object has a member field:

```
  header* head;
```

the pack and unpack functions for the packed class are:

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

Because head is passed as a pointer, if it is pointing to a `derivedheader` , the head field points to `derivedheader` when the packet is unpacked.

On the other hand, if the pack and unpack functions are written as follows, the `head` field points to a header (base class) when the packet is unpacked:

```
  virtual void do_pack(uvm_packer& p) const {
    p << *head << addr;
  }
  virtual void do_unpack(uvm_packer& p) {
    p >> *head >> addr;
  }
```

For a packed object to be successfully unpacked, the packing and unpacking functions must agree on the class names, the field types, and the field ordering.

Because the unpacking can occur in a different language than the packing, this generally means that the pack routine in one language must match the unpack routine in the other language for the class of the same name. The simplest way to ensure this is to make the class definitions in the two languages match, and both languages pack/unpack all of the fields. However, it is possible for the class definitions to differ, as long as the pack and unpack routines match.

The legal mapping between SystemVerilog and SystemC for the primitive data types is shown in the following table. For example, if your SystemC data object `packet` derives from `uvm_object` and contains two member fields of type `sc_logic` and `sc_int<64>` , the corresponding packet object in SystemVerilog should derive from `uvm_object` and contain two member fields of type `logic` and `bit[63:0]` , respectively.

| SystemVerilog Data Type | SystemC Data Type |
|---|---|
| bit | bool |
| logic | sc_logic |
| bit[N-1:0] | sc_bv<N>, sc_int<N>, sc_uint<N>, sc_bigint<N>, sc_biguint<N> |
| logic[N-1:0] | sc_lv<N> |
| byte | char |
| byte unsigned | unsigned char |
| shortint | short |
| shortint unsigned | unsigned short |
| int | int |
| int unsigned | unsigned int |
| longint | int64 |
| longint unsigned | uint64 |
| string | std::string, char* |
| dynamic array | std::vector |

If the class names on both sides do not match, and one side packs an `uvm_object` that the other side does not recognize, an error message is printed. The size of the two objects is also checked, and error messages are generated for too few or too many bits. Besides this, no other error checking is performed. For example, if one side packs a char and the other side is expecting an int, the result is a crash.

A field of type `enum` is passed across the language boundary as the basic non-enumeration type that it represents (usually `int` ). So, when packing and unpacking, you have to pack/unpack the corresponding basic data type, and then cast it to the `enum` type.

If you have a class with `enum` members, you can declare << and >> operators when you declare the enum inside your class. Then the pack/unpack functions can simply call << and >>, as usual. For example:

```
class packet : public uvm_object {
    ...
      enum state_t { OK=1, ERROR };

    friend uvm_packer& operator << (uvm_packer& p, state_t v) {
        p << (int)v;
        return p;
      }
     friend uvm_packer& operator >> (uvm_packer& p, state_t& v) {
       int i;
       p >> i;
       v = state_t(i);
       return p;
     }
     state_t state;
       ...
       virtual void do_pack(uvm_packer& p) const {
          p << data;
          p << state;
          p << address;
       }
     virtual void do_unpack(uvm_packer& p) {
       p >> data;
       p >> state;
       p >> address;
     }
  ...
};
```

# ML_TLM2_GP_BEGIN, ML_TLM_FIELD, and ML_TLM2_GP_END Macros

## Purpose

Create UVM-ML SystemC out-of-class serialization and de-serialization code for types derived from `tlm_generic_payload`.

## Definition

**ML_TLM2_GP_BEGIN (** *subclass-type* **)**
**ML_TLM2_FIELD(** *field-name* )
**ML_TLM2_GP_END(** *subclass-type* **)**

## Syntax Example

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

## Parameters

| | |
|---|---|
| *subclass-type* | The subclass for which you need a serializer. |
| *field-name* | The field name that has been added to the base class and needs to be serialized. |

## Description

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

### Example

Following is an example that shows how to write a serializer class for a type `trans` derived from `tlm_generic_payload` with an extra integer field of `f0`:

```
class trans: public tlm_generic_payload
{
public:
    trans():tlm_generic_payload() {}
    virtual ~trans() {}
    unsigned int f0;
};
ML_TLM2_GP_BEGIN(trans)

  ML_TLM2_FIELD(f0)
ML_TLM2_GP_END(trans)
```

# UVM-ML SystemC Instantiation of Foreign Child Components

## uvm_ml_create_component()

### Purpose

Instantiate a child component that is implemented in a different framework (a different language).

### Definition

child_component_proxy * **uvm_ml_create_component(**
  const std::string & *target-frmw-indicator* **,**
  const std::string & *component-type-name* **,**
  const std::string & *instance-name* **,**
  const uvm_component * *parent* **);**

## Syntax Example

```
child_component_proxy * svtop;
   svtop = uvm_ml_create_component("SV", "sv_uvc", "sv_uvc1", this);
```

## Parameters

| Parameter | Description |
|---|---|
| *target-frmw-indicator* | Identifies the framework that implements the foreign child component. For example: <br>• "sv" when instantiating a UVM-SystemVerilog component.<br>• "e" when instantiating an ***e*** component. |
| *component-type-name* | The type of the foreign child component, as defined in the component's framework. |
| *instance-name* | The name of the top instance in the remote subtree. |
| *parent* | The pointer to the parent instance in the current framework. Use **this** to indicate the current component. |

## Description

Instantiates a child component in a different framework from the current one when you are creating unified hierarchy .

This function returns an object of type `uvm_ml::child_component_proxy` , which is defined in the UVM-ML SystemC header file.

The class `child_component_proxy` is derived from the base class `uvm_component` . The actual return value type can be defined in the user code using the base class `uvm_component` . For example:

```
uvm_component my_uvc_env;
...
my_uvc_env = uvm_ml_create_component(....);
```

Declaring the return component using the abstract base class `uvm_component` can help with future reuse, re-factoring, or refinement of the implementation. In such cases, the actual component might be once implemented as a foreign component in one configuration and as a native SystemC component in another configuration. Using a common base class allows for all configurations (with no changes required).

⚠ The instantiation by a foreign language takes place in simulation time, so in cases where SystemC is instantiating an ***e*** unit, features that require stub code will not work without special handling. Such features include method ports, Verilog wire access, Verilog/VHDL access, part select, and other interface constructs. For how to handle this situation, see Specman Stub Generation for Sub-Trees in a Unified Hierarchy .

## Example

The following example creates a SystemVerilog sub-tree of type `svtop` with instance name `sv_uvc1` :

```
child_component_proxy * sv_uvc1;
void phase_started(uvm_phase* phase) {
    if (phase->get_name() == "build") {
        sv_uvc1 = uvm_ml_create_component("SV", "sv_uvc", "sv_uvc1", this);
    }
}
```

# UVM-ML SystemC Configuration

This section contains the following:

- Functions uvm::set_config_*()
- Functions uvm::get_config_*()

# Functions uvm::set_config_*()

## Purpose

Sets UVM configuration values to be propagated to all frameworks.

## Definition

```
template <typename T > void  set_config_int(
   const std::string& inst-name ,
   const std::string& field-name ,
   const T& value
);
void set_config_string(
   const std::string& inst-name ,
   const std::string& field-name ,
   const std::string&  value
);
void set_config_object(
   const std::string& inst-name ,
   const std::string& field-name ,
   uvm_object* value ,
   bool clone = true
);
```

## Arguments and Parameters

| **Argument or Parameter** | **Description** |
|---|---|
| *T* | The value type . See UVM-ML SystemC Adapter Basics for supported types. |
| *inst-name* | The relative path of the instance for which a configuration value is being set. You can use the " `*` " wildcard. |
| *field-name* | The name of the configuration field being set. In a unified hierarchy, the value set for this field is propagated to all possible targets with a matching *inst-name.field-name* . |
| *value* | The value to be assigned to the given field (integral value, string value, or serializable object value). |
| **Note** : The Boolean argument 'clone' for `set_config_object()` is currently not supported. As a result, `get_config_object()` always returns a new copy of the object, and you are responsible for deleting it after usage. | |

## Description

Set configuration values. These functions can be used in a single-language UVM-SystemC environment or in a UVM-ML environment. When in a UVM-ML environment, the configuration data is propagated also to other frameworks.

Setting a configuration value that affects the construction of the testbench should be done before building the sub-components (in a constructor or a build phase of the parent component).

### Notes

- All integral values are converted by the adapter to `sc_bv<4096>` and then passed to other frameworks as a raw bit vector. This way other frameworks can cast the value to their local type provided that it is compatible with the original type.
- In a unified hierarchy, setting configuration values is supported only for foreign child components. It is not possible, for example, to configure a foreign sibling component.
- If there is a typo in the field name, the configuration setting is ignored (and there is no warning message).

### Example

The following example sets the “address” field in all child components whose instance name ends with "producer":

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

### See Also

- UVM-ML Configuration in a Unified Hierarchy

## Functions uvm::get_config_*()

### Purpose

Get UVM configuration values. In a unified hierarchy, these values can be propagated from other frameworks.

## Definition

template <typename *T* > bool **get_config_int(**
  const std::string& *field-name* **,**
  T& *value-field-name*
**);**
bool **get_config_string(**
  const std::string& *field-name* **,**
  std::string& *value-field-name*
**);**
bool **get_config_object(**
  const std::string& *field-name* **,**
  uvm_object*& *value-field-name* **,**
  bool clone = true
**);**

## Arguments and Parameters

| **Argument or Parameter** | **Description** |
|---|---|
| *T* | The value type . See UVM-ML SystemC Adapter Basics for supported types. |
| *field-name* | The name of the configuration field whose value is being retrieved. |
| *value-field-name* | The reference to the field into which the value is retrieved. |
| **Note** : The Boolean argument for `set_config_object()` is currently not supported. As a result, `get_config_object()` always returns a new copy of the object, and you are responsible for deleting it after usage. | |

## Description

Retrieve configuration values. These functions can be used in a single-language UVM-SystemC environment or in a UVM-ML environment. When in a UVM-ML environment, the configuration data can be retrieved from other frameworks.

## Example

The following example retrieves the value of “address”:

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

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

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

Note that the dynamic cast can fail if the type of the value set does not match the type provided as the get...() argument.

## See Also

- UVM-ML Configuration in a Unified Hierarchy

# UVM-ML SystemC Debugging Utilities

uvm_ml_execute_command()

## Purpose

Activate a UVM-ML command procedurally from a user code

## Definition

```
int uvm_ml_execute_command(
   const std::string &  command ,
   sc_severity error_severity = SC_ERROR );
```

## Syntax Example

```
int command_result = uvm_ml_execute_command("uvm_ml_trace register -on", SC_FATAL);
```

## Parameters

| Parameter | Description |
|---|---|
| *command* | A UVM-ML command in the same format as documented for the interactive invocation |
| *error_severity* | Reaction upon an error (if any) during execution of the command. This argument is of the standard SystemC type sc_severity.<br>The SC_FATAL argument causes invocation of  SC_REPORT_FATAL() in case of an error, SC_ERROR - SC_REPORT_ERROR(). |
| *return value* | 1 - if the command was executed successfully, 0 - if the command failed, (-1) - if the command was not recognized |

## Description

Sometimes the user can not use an interactive prompt to provide a debugging command. One example of such situation is when it's necessary to to debug
activities happening in the sc_main() function. Sc_main() is executed very early, before the user has a chance to supply an interactive command. In such case,
the user can invoke uvm_ml_execute_command() in the beginning of sc_main() before any relevant ML actions (e.g. before uvm_ml_register_target/initiator().

The SC UVM-ML adapter passes this command for actual execution to the UVM-ML backplane - hence the command can affect all integrated frameworks and not only
SystemC.

## Example

The following example shows invocation of uvm_ml_execute_command from sc_main():

```
int sc_main(int argc, char* argv[]) {
        my_env_t my_env_inst("my_env");

 void(uvm_ml_execute_command("uvm_ml_trace_register -on"));

        ML_TLM2_REGISTER_TARGET(my_env, tlm_generic_payload, tsocket, 64);

        ...
    }
}
```

## Limitations

Currently, only *uvm_ml trace_register* command is supported. Other UVM-ML debugging commands cannot be activated procedurally yet.

4

# UVM-ML e Interface

This section describes the following:

## UVM-ML e Adapter Basics

This section describes the following:

⚠ You do not need to explicitly import or include any UVM-ML package or adapter or library for use with the ***e*** language. All such components of the UVM-ML offering are implicitly available for ***e*** .

### UVM-ML Framework Identifiers

Framework identifiers are used to identify the framework, for example, when you instantiate a child component that belongs to another framework.

The framework identifiers for the frameworks released with UVM-ML are as follows:

- "sv" or "uvmsv" for the UVM-SV adapter

- "sc" or "uvmsc" for any UVM-SC adapter
- "e" for the UVM- ***e*** adapter

⚠ All framework identifiers are case-insensitive.

## Data Types Supported for UVM-ML *e* Configuration and TLM Communication

UVM-ML ***e*** supports structs for multi-language configuration values or TLM transactions. The struct fields must be of the following types:

- scalar types and subtypes, including enumerated scalar types
- struct types
- string type
- lists of any of the above types

In addition, raw integer values and string values are supported for passing as configuration values.

### See Also

- Data Communication in a UVM-ML Environment
- UVM-ML e TLM Interface

# UVM-ML e Instantiation of Foreign Child Components

## child_component_proxy is instance

### Purpose

Define a unit instance field of type `child_component_proxy` to enable instantiation of a foreign framework component.

### Syntax

*instance-name* **: child_component_proxy is instance;**
  **keep** *instance-name.* **type-name** **==** *foreign-type-identifier* **;**

### Syntax Example

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

## Parameters

| **Parameter** | **Description** |
|---|---|
| *instance-name* | The instance name of the foreign child component and of the ***e*** proxy unit (both receive the same instance name, although they are in different frameworks). This name can be used, for example, to connect its port or configure its field. |
| *foreign-type-identifier* | The type of the foreign child (the top instance of the foreign sub-tree), as defined in the framework of the foreign sub-tree. Must be prepended with the framework identifier (for example, "SV:svtop" or "SC:sctop"). |

## Description

To instantiate a child component in a different  framework while in the ***e*** framework, you :

1. Define an ***e*** instance of type `child_component_proxy` , which acts as the "proxy" for the foreign child.
2. Constrain the ***e*** instance's `type_name` field to the foreign child instance name.

Note that you must use the Specman IntelliGen generator with UVM-ML. (You cannot use Pgen.)

## Example

```
unit ubus_env_proxy {
    // Foreign (SV) child
    uvc_top: child_component_proxy is instance;
    keep uvc_top.type_name== "SV:ml_ubus_env" ;
    // Configure UBUS
    keep uvm_config_set ( "uvc_top" , "slave0_max" , 16 'h7fff); // slave 0 address space
    keep uvm_config_set ( "uvc_top" , "slave1_max" , 16 'hffff); // slave 1 address space
};
```

## See Also

- UVM-ML Hierarchy
- "Instantiating a SystemVerilog Component Within an e Unit" in the *UVM-ML OA User Guide*

# UVM-ML e Configuration

This section describes the following:

- keep uvm_config_set()
- keep uvm_config_get()
- uvm_ml_packed_struct

## keep uvm_config_set()

### Purpose

Sets UVM-ML configuration values to be propagated to all frameworks.

### Category

Generation constraint

### Syntax

**keep uvm_config_set(** *inst-name* **,** *field-name* **,** *value* **);**

### Syntax Example

```
extend testbench {

    // configure UBUS.
    // The following configures all components whose path contains "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
};
```

## Parameters

| Parameter | Description |
|---|---|
| *inst-name* | The relative path of the instance for which a configuration value is being set. You can use the " `*` " wildcard. |
| *field-name* | The name of the configuration field being set. In a unified hierarchy, the value set for this field is propagated to all possible targets with a matching *inst-name/field-name* . |
| *value* | The value to be assigned to the given field (integral value, string value, or serializable object value). |

## Description

Sets UVM-ML configuration values.

⚠ Values set with `uvm_config_set()` cannot depend on generated values in sub-units because they are generated after `uvm_config_set()` is executed.

## Notes

- If there is a typo in the unit or field name, the configuration setting is ignored (and there is no warning message).
- `uvm_config_set()` and `uvm_config_get()` currently cannot be used in a procedural context.

## Example

See "Instantiating a SystemVerilog Component Within an e Unit" in the *UVM-ML OA User Guide.*

## See Also

- UVM-ML Configuration in a Unified Hierarchy
- "Data Types Supported for UVM-ML e Configuration and TLM Communication" in UVM-ML e Adapter Basics
- "trace uvm_config" in Specman Debugging Commands for Use in a UVM-ML Environment

# keep uvm_config_get()

## Purpose

Gets configuration values from the configuration database. Used to retrieve ***e*** configuration values that are set natively or by another framework.

## Category

Generation constraint

## Syntax

**keep** [ **soft** ] **uvm_config_get(** *field-or-attribute* **);**

## Syntax Example

```
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);
};
```

## Parameters

| Parameter | Description |
|---|---|
| *field-or-attribute* | The name of the field in which the config value is stored.<br>Besides data fields, the *value-field* can contain the value of a unit attribute: Examples:<br>`keep soft uvm_config_get(agent());`<br>`keep soft uvm_config_get(hdl_path());` |

## Description

UVM- ***e*** uses the configuration values during the generation of the testbench, which is a declarative process. Therefore, the syntax `keep uvm_config_get()` is declarative.

## uvm_config_get() Example

```
unit my_agent like uvm_agent {
    conf: my_config_object;
    keep uvm_config_get(conf);
    name: string;
    keep soft uvm_config_get(name);
    address: int;
    keep check_active() => uvm_config_get(address);
    check_active(): bool is {
        result = (me.active_passive == ACTIVE);
    };
};
```

## Notes Regarding Unit Attributes

- Using `uvm_config_get()` for unit attributes makes sense when the unit is instantiated in a foreign framework component.
- The predefined ***e*** unit base classes `uvm_env` and `uvm_agent` already contain the soft constraints:

```
keep soft uvm_config_get(agent());
keep soft uvm_config_get(hdl_path());
```

  Thus, you not need to add them if the ***e*** top subtree unit inherits from one of those base units.

## See Also

- UVM-ML Configuration in a Unified Hierarchy
- "trace uvm_config" in Specman Debugging Commands for Use in a UVM-ML Environment

# uvm_ml_packed_struct

## Purpose

Force an ***e*** struct to be mapped to SystemVerilog packed struct when it is being passed during configuration.

## Category

Statement

## Syntax

**uvm_ml_packed_struct** *struct-type-name* **;**

## Syntax Example

```
uvm_ml_packed_struct packet ;
```

## Parameters

| Parameter | Description |
|---|---|
| *struct-type-name* | The ***e*** struct that is to be mapped to a SystemVerilog packed struct. |

## Description

By default, an ***e*** struct is mapped to a foreign language class. However, you can use `uvm_ml_packed_struct` to force an ***e*** struct to be mapped to a SystemVerilog packed struct when it is being passed during configuration.

## Notes

- Units are not supported
- Only physical fields are passed.
- Only structs with statically sized fields are supported. This means:
    - Numeric types.
    - Structs with scalar fields with no like inheritance or when subtypes.
    - Structs with physical fields of type `list of scalar` for which the list size is explicitly defined in the field declaration as follows:

      ```
      field-name [ list-size ]: list of scalar-element-type
      ```

## Example

The following example consists of two e files: dt.e, which defines a struct named pdata that is mapped to a SystemVerilog packed struct, and top.e, which the pdata struct as a configuration setting to a SystemVerilog component.

dt.e File:

```
<'
struct data {
    % addr:int;
```

```
    % trailer:int;
    % txt:string;
};

struct pdata {
    % data:int;
    % addr:uint(bits:4);
    % payload:int;
};

// Marks pdata is mapped to a SV packed struct
uvm_ml_packed_struct pdata;
'>
```

## top.e File:

```
<'
import dt;
unit u {

     my_sv_child: any_unit is instance;
     keep soft my_sv_child == NULL;

     d : data;
     keep d.addr == 10;
     keep d.trailer == 20;
     keep d.txt == "config object msg";

     pd : pdata;
     keep pd.data == 30;
     keep pd.addr == 4;
     keep pd.payload == 50;

     keep uvm_config_set("my_sv_child","conf_data",d);
     keep uvm_config_set("my_sv_child","conf_pdata",pd);
    ...

};

extend u {
    keep my_sv_child != NULL;
    keep type my_sv_child is a child_component_proxy;
    keep my_sv_child.type_name == "SV:test";
    };

    >'
```

## See Also

- keep uvm_config_set()

# UVM-ML e Type Mapping

## uvm_ml_type_match

### Purpose

Map class (struct) names across frameworks for TLM transactions.

### Syntax

**uvm_ml_type_match** *type1-identifier* *type2-identifier* **;**

### Syntax Example

```
    uvm_ml_type_match "e:cdn_xbus::MASTER xbus_trans_s" "sv:xbus_trans_s";
```

## Parameters

| Parameter | Description |
|---|---|
| *type1-identifier, type2-identifier* | The type names for the two types to be mapped. The type names:<br>● Can be short names or fully qualified (including a namespace, package name, and so on).<br>● Must start with the framework identifier (for example, e, sv, sc).<br>● Can appear in any order.<br>Examples:<br>● "sv:my_pkg:sv_packet"<br>● "e:main:e_packet" |

## Description

This function is used to de-serialize objects. UVM-ML supports polymorphic transactions; thus, the multi-language adapter on the consumer-side needs to create a class object that matches the class of the actual argument on the producer side, at runtime. By default, the mapping between such classes assumes that the class names are the same. This function establishes correspondence between two types whose names are different.

You need to map type names only once, in one of the supported framework languages.

## See Also

- Data Communication in a UVM-ML Environment
- "Type Mapping" in the *UVM-ML OA User Guide*

# UVM-ML e TLM Interface

This section contains the following:

- uvm_ml.connect_names()

- any_tlm_socket.turn_transaction_mapping_off()
- uvm_ml_config.tlm_pass_field()

# uvm_ml.connect_names()

## Purpose

Connect  any ports or sockets in any framework  in a UVM-ML environment.

## Category

Method of the predefined ***e*** singleton struct `uvm_ml`

## Syntax

**uvm_ml.connect_names(** *initiator_path* : string, *target_path* : string **)** : bool **;**

## Syntax Example

```
res = uvm_ml.connect_names("sys.my_producer.my_port",
append(sc_consumer,"my_export"));
```

(See example description below.)

## Parameters

| | |
|---|---|
| *initiator_path* | The full UVM-ML path to the TLM1 port or TLM2 initiator socket.<br>In case of the hierarchical construction, it can also be a name of an export (e.g. in SV, e) if it is being connected to an implementation. |
| *target_path* | The full UVM-ML path to the TLM1 port or TLM2 initiator socket. |

## Description

Use this method to connect UVM-ML TLM ports. The return value is TRUE if the connection is successful.

Connecting the ports needs to be done in one language only and can be done from any framework. For example, you could connect SystemVerilog/SystemC ports from an ***e*** test file.

## Example

The following example connects an ***e*** port named "my_port" to an SystemC export named "my_export" (sc_consumer is the path to the SystemC consumer component):

```
res = uvm_ml.connect_names("sys.my_producer.my_port",
append(sc_consumer,"my_export"));
```

## See Also

- UVM-ML TLM Communication

# any_tlm_socket.turn_transaction_mapping_off()

## Purpose

Turns off transaction mapping for specified TLM2 socket(s).

## Category

Predefined method for TLM2 sockets

## Syntax

**any_tlm_socket.turn_transaction_mapping_off( );**

## Syntax Example

```
tlm_init.turn_transaction_mapping_off();
```

## Description

In a multi-language environment, TLM2 transactions are passed by copy, as a serialized bit stream. In this environment, ensuring that a returned transaction is the same object as the object that was sent (a process called "transaction mapping") results in runtime performance and memory costs. By default, transaction mapping is on for all TLM2 target sockets.

At times, you might consider the cost unjustified, for example because certain sockets are used only on a forward path. In such cases, you can turn off transaction mapping for the specified socket(s) with `tlm_transaction_mapping_off()` . The result will be that any TLM2 transaction on the backward path is carried in different object than the object sent on the forward path.

## See Also

- "More Information About the Multi-Language TLM2 Interface" in UVM-ML TLM Communication

# uvm_ml_config.tlm_pass_field()

## Purpose

Control which fields of a struct will be passed via TLM communication or ***e*** configuration.

## Category

Predefined method

## Syntax

**uvm_ml_config.tlm_pass_field(** *field* **: rf_field): bool**

## Syntax Example

```
extend uvm_ml_config {
    tlm_pass_field(f:rf_field) : bool is also {
        if ( f.get_declaring_struct().get_name() == "packet" ) {
            var n : string = f.get_name();
            if ( n == "name" or n == "id") {return TRUE;};
        };
    };
};
```

## Parameters

| Parameter | Description |
|---|---|
| *field* | Name of the ***e*** field. |

## Description

By default, only physical fields of ***e*** structs are passed in multi-language configuration or TLM1/TLM2 transactions. To control the set of fields being passed, you can extend the method `uvm_ml_config.tlm_pass_field()`.

This method receives one parameter of type rf_field, and returns TRUE if this field is to be passed or FALSE otherwise.

## Example

In this example, first a struct named “packet” is defined with three fields named id, name, and data and a pointer to driver. Only the field data is physical.

```
struct packet {
    id : uint;
    name : string;
    %data : list of byte;
    !driver: xbus_driver;
};
```

The following code extends the `tlm_pass_field()` method to ensure that the fields `id` and `name` are passed as well as the physical field `data` :

```
extend uvm_ml_config {
    tlm_pass_field(f:rf_field) : bool is also {
        if ( f.get_declaring_struct().get_name() == "packet" ) {
            var n : string = f.get_name();
            if ( n == "name" or n == "id") {return TRUE;};
        };
    };
};
```

## See Also

- The description of `tlm_pass_field()` in "Type Mapping" in the *UVM-ML OA User Guide*
- UVM-ML TLM Communication

# Specman Stub Generation for Sub-Trees in a Unified Hierarchy

If you have an ***e*** unit instantiated under a foreign component, you might need to create a stub unit.

Certain ***e*** declarations require generation of auxiliary code in a file, commonly called a "stub file". The problem is that the unified multi-language testbench hierarchy is not known when the stub file is normally generated (either manually or by irun). Thus, the necessary stub code is not automatically generated.

In this case, you must generate the stub code manually. To do so, you:

1. Declare a special "stub-unit" that mimics the future to-be-instantiated parent-proxy unit
2. Set the special "stub-unit" to have the intended unified-hierarchy path
3. Tell Specman to generate the required stub code (either by running Incisive irun or with the Specman `write stub` command).

## When Stub Code Is Required

The ***e*** declarations that require stub code include:

- Legacy Verilog statements ( `verilog code` , `verilog function` , `verilog import` , `verilog task` , `verilog time, verilog variable` ).
- All external ***e*** ports that are documented as requiring stub code. To know if a given port requires stub code, you can use the ***e*** method *`port`* `.is_stub_required()`.

Note that TLM1 ports and TLM2 sockets do **not** require stub code.

## Example

The following complete example illustrates the use of the ***e*** methods that generate the required stub code.

⚠ You could also use the convenience macro `uvm_ml_stub_unit` to execute the ***e*** methods illustrated in this example. See uvm_ml_stub_unit Macro for more information.

### UVM-ML SystemVerilog Code:

In the following example, the SystemVerilog code instantiates an ***e*** unit of type `my_unit` with the instance name of `u1` . `my_unit` contains a port that drives a Verilog wire in the DUT and therefore requires a stub. The example uses the UVM-ML configuration mechanism to set the `hdl_path()` and `agent()` unit attributes. Without setting these fields, the example will fail in simulation time because the unit instance does not exist at stub time, so no stub code is generated for the port `pw` .

```
class env extends uvm_env;
    uvm_component u1; // e sub tree;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        // setting the hdl_path() attribute of e unit
        uvm_config_string::set(this,"u1","hdl_path()","topmodule");
        // setting the agent() attribute of the e unit
        uvm_config_string::set(this,"u1","agent()","SV");
        // create ML parent proxy in e
        u1 = uvm_ml_create_component("e", "my_unit", "u1", this);
    endfunction
```

```
endclass
```

## UVM-ML *e* Code:

The e code below illustrates the use of the two predefined methods used to generate the e stub code: `sys.pre_stub_generation()` and `uvm_ml_create_stub_unit()`.

`sys.pre_stub_generation()` is called by Specman automatically during stub generation after `sys.setup()` and before the generation phase. Within this method, you can create the ***e*** parent proxy by calling the predefined global method `uvm_ml_create_stub_unit()`. This method creates a parent proxy of the user-defined type (in the example below “my_parent_proxy”). The second argument of this method must match the actual instance name (that is, the UVM-SystemVerilog full name) of the parent foreign component (in the example below “uvm_test_top.my_env”).

The parent proxy unit that inherits from `uvm_ml::parent_component_proxy` (in this example “my_stub_unit”) instantiates the actual unit (or units, in this example “u1”) and sets its configurations. In the example below, unit "my_stub_unit" instantiates “u1” of type "my_unit", which is the actual unit that is instantiated by `uvm_ml_create_component()` in SystemVerilog. The example also replicates the configuration settings done in SystemVerilog to the parent proxy. Thus, the `hdl_path()` and `agent()` attribute will be available in stub time to create the right entries in the stub file.

```
extend sys {
    pre_stub_generation() is also {
        uvm_ml_create_stub_unit(”my_stub_unit”,”uvm_test_top.my_env”);
    };
};
unit my_stub_unit like uvm_ml::parent_component_proxy {
    u1: my_unit is instance;
    keep uvm_config_set("u1","agent()","SV");
    keep uvm_config_set("u1","hdl_path()","topmodule");
};
unit my_unit like uvm_agent {
    keep soft uvm_config_get(hdl_path());
    keep soft uvm_config_get(agent());
    pw : out simple_port of int is instance;
    keep pw.hdl_path() == "my_wire";
    keep pw.verilog_wire() == TRUE; // This port requires stub code
};
```

## In this Section

The following sections describe the ***e*** methods you use to generate the stub code for an ***e*** unit instantiated under a foreign component:

- uvm_ml_create_stub_unit()
- sys.pre_stub_generation()
- uvm_ml_stub_unit Macro

## See Also

- UVM-ML Hierarchy
- "Stub Generation for an e Sub-Tree" in the *UVM-ML OA User Guide*
- The description of the 'write stubs' command in the *Specman Command Reference*

# uvm_ml_create_stub_unit()

## Purpose

Create a unit of a user-defined parent proxy type.

## Category

Predefined global method.

## Syntax

**uvm_ml_create_stub_unit(** *parent-proxy-type_name* : string, *target-e-path* : string **);**

## Syntax Example

```
    uvm_ml_create_stub_unit(“stub_unit_for_xbus”,”uvm_test_top.system_env.xbus_evc”) ;
```

## Parameters

| Parameter | Description |
|---|---|
| *parent-proxy-type-name* | The type of the stub unit. The type must be defined and inherit from the predefined type `uvm_ml::parent_component_proxy` or an error is generated. |
| *target-e-path* | The ***e*** path that the created parent proxy should have at simulation time. |

## Description

This method creates a stub unit instance of type `parent_proxy_type_name` . You must call this method from sys.pre_stub_generation() . Calling it from any other place results in an error message.

The *target_e_path* should be the ***e*** path that the instantiated unit will have at simulation time (that is, when the foreign framework creates it). Therefore, this ***e*** path needs to be be unique (so you cannot create multiple unit instances with the same ***e*** path).

## See Also

- For more information and a complete example, see "Specman Stub Generation for Sub-Trees in a Unified Hierarchy" in the *UVM-ML OA User Guide.*

# sys.pre_stub_generation()

## Purpose

A hook method used to create stub code for ***e*** units instantiated under a foreign language.

## Category

Predefined method of `sys`

## Syntax

**sys.pre_stub_generation()**

## Syntax Example

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

## Description

Use this method to create ***e*** parent proxies during stub writing when ***e*** units are instantiated under a foreign language

When the stub code is generated, the `sys.pre_stub_generation()` method is called after `setup` phase and before `pre_generate()` is called.

When a UVM-ML environment contains ***e*** sub-trees that are not instantiated under **sys** (for example, multiple ***e*** UVCs), `pre_stub_generation()` should include a separate call to `uvm_ml_create_stub_unit()` for each sub-tree.

## See Also

- For more information and a complete example, see "Specman Stub Generation for Sub-Trees in a Unified Hierarchy" in the *UVM-ML OA User Guide .*

# uvm_ml_stub_unit Macro

## Purpose

Create an ***e*** unit instantiated in a parent proxy for stub writing with optional unit attributes.

## Category

Predefined macro statement

## Syntax

**uvm_ml_stub_unit** *full-UVM-path* **using**
**type=** " *unit-type-name* "
[ **, hdl_path=** " *hdl-path* "]
[ **, agent=** " *agent* "]
[ **, external_uvm_path=** " *external-UVM-path* "] **;**

## Syntax Example

```
uvm_ml_stub_unit uvm_test_top.top_env.uvc_env \
    using type="uvc_env_t", agent="SV", hdl_path="dut_top",
external_uvm_path="uvm_test_top.top_env";
```

## Parameters

| | |
|---|---|
| *full-UVM-path* | The full UVM name (instance path) of the unit as it will be created at runtime in the target ML environment. |
| *unit-type-name* | The type name of the instantiated unit. |
| *hdl-path* | The value of the `hdl_path()` attribute this unit will have. |
| *agent* | The value of the `agent()` attribute this unit will have. |
| *external-UVM-path* | The value of the `external_uvm_path()` attribute this unit will have. This is an ***e*** method port attribute. |

## Description

The `uvm_ml_stub_unit` macro provides an alternative to using `pre_stub_generation()` and `uvm_ml_create_stub_unit()` for generating stub code for an e unit instantiated under a SystemVerilog component.

## See Also

- Specman Stub Generation for Sub-Trees in a Unified Hierarchy
- "Stub Generation for an e Sub-Tree" in the *UVM-ML OA User Guide*

# 5 UVM-ML Commands

This section contains the following:

## irun -uvmtest -uvmtop

The IES tools irun and ncsim provide an alternative method to using the UVM-ML SystemVerilog task `uvm_ml_run_test()` . When you start the simulation with irun, you can define the top and test components using command-line declarations `-uvmtest` and `-uvmtop` . For example:

```
irun -uvmtest SV:my_test -uvmtop SC:sctop …
```

- `-uvmtest` declares a topmost test component to be instantiated and built at the start of the simulation. This is the command line equivalent of the `test` option for `uvm_ml_run_test()` .
- `-uvmtop` declares an additional top-level component to be instantiated and built at the start of the simulation. This is the command line equivalent of the `tops` option for `uvm_ml_run_test()` .
    - One can have multiple uvmtop declarations on the command line.
    - The instance name of `uvmtop` components can be provided in addition to the type name, for example:

      ```
      -uvmtop SC:sc_ref_mod_t:ref_mod_top_instance_name
      ```

      The default instance name is the same as the type name provided.

## Notes:

- When you specify `-uvmtest` and `-uvmtop`, you do not need to include the `irun -ml_uvm` option.
- Do not use `-sctop` to identify top SystemVerilog components; it creates a static module instance that does not participate in the quasi-static phases.
- When the test is invoked using -uvmtest, UVM-ML takes care of the elaboration phases just before the execution of any DUT processes, including the SystemVerilog initial blocks. If you need to configure some build-time settings from a SystemVerilog module, you should make them in a static initialization function call as shown below:

```
module topmodule;
    ex_if my_if();
     function bit set_if(virtual ex_if vif);
          uvm_config_db#(virtual ex_if)::set(null, "*", "my_vif", vif);
          return 1;
     endfunction
     bit res = set_if(my_if);
...
```

## See Also

- UVM-ML SystemVerilog Start of Test
- "irun Options Used with UVM-ML" in "Running UVM-ML with Incisive (IES)" in the *UVM-ML OA User Guide*

# Incisive Debugging Commands for Use in a UVM-ML Environment

The Incisive commands in this section can help you debug your multi-language environment. The debug commands presented in this chapter can be activated in two ways:

- As IES Tcl commands- when using UVM-SV 1.1d, these commands are sourced by irun automatically from $UVM_ML_HOME/ml/frameworks/uvm/sv/1.1d-ml/uvm_lib/uvm_sv/files/tcl/uvm_sim.tcl.  Currently when using UVM-SV 1.2, you must source this tcl file explicitly by specifying -input ${UVM_ML_HOME}/ml/facilities/debug/uvm_ml_debug.tcl on the command line.

- As Specman commands that can be activated from Specman prompt with any one of the supported simulators.

- uvm_ml phase Command
- uvm_ml print_connections Command
- uvm_ml print_tree Command
- uvm_ml trace_register_tlm Command

## See Also

- "Debugging in Multi Language Environment" in the *UVM-ML OA User Guide*

# uvm_ml phase Command

## Purpose

Enables you to stop at the beginning or at the end of a phase .

## Syntax

**uvm_ml phase**
  **-stop_at ( -begin** *phase-name* | **-end** *phase-name* | **-build_done )**
 | **-remove_stop ( -begin** *phase-name* | **-end** *phase-name* | **-all )**
 | **-get**
 | **-list_stop**
 | **-run** *phase-name*
 | **-h** [ **elp** ]

## Parameters

| Parameter | Description |
|---|---|
| *phase-name* | The phase name is a string value. At runtime, UVM-ML compares the actual phase name with the name provided in the request.<br>If the phase controller is UVM-SystemVerilog or the default internal service provider, the supported phases are those listed in "The Phasing Controller" in Synchronized Phasing in a UVM-ML Environment . |
| **-stop_at** | **-begin** *phase_name* sets the callback for the beginning of the phase.<br>**-end** *phase_name* sets the callback for the end of the phase.<br>**-build_done** sets a callback when the primary environment build is complete. This is equivalent to setting a stop at the end of the build phase, at which time the verification environment should be completely built. |
| **-remove_stop** | **- begin** *phase_name* cancels the breakpoint at the beginning of the phase.<br>**- end** *phase_name* cancels the breakpoint at the end of the phase.<br>**- all** cancels all active UVM-ML breakpoints. |
| **-get** | Get the name of the current phase. |
| **-list_stop** | List all the active UVM-ML breakpoints. |
| **-run** *phase-name* | Run to *phase_name* . **-run** is similar to **stop_at -begin .** After setting the breakpoint at the beginning of a phase, it resumes the session.<br>Note, if you use this parameter from Specman, the breakpoint will be set however the session will not be resumed. |
| **-help** | Lists descriptions of all parameters. |

## Description

Enables you to stop at the beginning or at the end of a phase, to allow examination of the simulated environment. The most common usage of this command is to set a break point for a specific phase (either the start of the end of the phase).

## Notes

- If a stop request relates to a phase that is already done, the request does not get executed.
- Breakpoints remain active until they are removed or the session terminates.
- The stop requests are preserved in reset.

## Limitations

- The Tcl commands are currently supported only by IES (however, the commands can be activated from Specman with any one of the supported simulators).
- Currently, only the UVM-ML SystemVerilog framework can stop the session. If the environment does not contain a UVM-ML SystemVerilog framework, the session cannot be stopped by this command.

## Examples

Setting a breakpoint before the build phase:

```
% uvm_ml phase -stop_at -begin build
```

Canceling a breakpoint with `remove_stop` :

```
% uvm_ml phase -remove_stop -begin build
% uvm_ml phase - remove_stop -end build
% uvm_ml phase - remove_stop -end connect
```

Listing the pending breakpoint requests:

```
% uvm_ml phase -list_stop
UVM-ML: Stop #1 build begin
UVM-ML: Stop #2 connect end
```

# uvm_ml print_connections Command

## Purpose

List multi-language connections of TLM ports and sockets.

## Syntax

**uvm_ml print_connections** [[ **-t** [ **ype** ]] [ **-r** [ **oot** ] *path* ]

**uvm_ml print_connections -h** [ **elp** ]

## Parameters

| Parameter | Description |
|---|---|
| **-type** | Prints type names, if this information is accessible |
| **-root** *path* | Prints the full path to the root node where the display starts. |
| **-help** | Lists descriptions of all parameters. |

## Description

Lists the multi-language connections in the environment. You can request a partial listing by specifying the root at which the list is to start.

The text output includes arrows that indicate the direction from initiator to target.

## Examples

Print the connections at the end of the connect phase:

```
 % uvm_ml print_connections
List of UVM-ML connections:
Specman:sys.e_env.producer.nb_socket
    -> UVM SV:uvm_test_top.sv_env.consumer_1.nb_target_socket
```

...

Print the connections with the type, under a given root:

```
% uvm_ml print_connections -t -r uvm_test_top.sv_env.consumer_1.b_target_socket
List of UVM-ML connections:
UVM SV:uvm_test_top.sv_env.consumer_1.b_target_socket [uvm_tlm_b_target_socket]
  <- Specman:sys.e_env.producer.b_socket [tlm_initiator_socket of
tlm_generic_payload]
```

## Note

- The SystemC adapter can not extract type information about the port types because of the native language limitations. Thus, the type name is left empty for SystemC ports and sockets.

## uvm_ml print_tree Command

### Purpose

Print hierarchical tree(s) in the multi-language environment. For use after the build phase is done.

### Syntax

**uvm_ml print_tree**
[- **r** [ **oot** ] *root_path* ]

[- **d** [ **epth** ] *depth* | **all** ]
[- **p** [ **orts** ]]
[- **t** [ **ype** ]
[- **s** [ **hort** [ **_path** [ **s** ]]]]

**uvm_ml print_tree -h** [ **elp** ]

## Parameters

| Parameter | Description |
|---|---|
| - **r oot** *root_path* | Prints the tree that starts at the specified root. The default is to print the entire environment. |
| - **d epth** *depth* \| **all** | ● *depth* specifies how many levels to show. ● **all** lists all levels. The default is three levels. |
| - **p orts** | Displays ports as child components. For a more complete display of port connections, use the `uvm_ml_print_connections` tcl command. |
| - **t ype** | Displays the type name after the component name. |
| - **s hort _path s** | Displays the component name only. |
| **-help** | Lists descriptions of all parameters. |

## Description

If you specify a root component, this command prints the hierarchical tree that starts with that component. If you do not specify a root component, this command prints the hierarchy for the entire environment. In a unified hierarchy, this means printing the entire tree. In a side-by-side hierarchy, this means printing all subtrees in the environment.

The hierarchy that is printed contains the building blocks of the verification environment. In SystemVerilog and SystemC, these are the classes derived from `uvm_component` . In ***e*** , these are units. If ports are added using the `-ports` parameter, all TLM ports and sockets are added.

## Examples

Print the top 3 levels of a multi-langauge environment at the end of the build phase:

```
% uvm_ml print_tree

sv: uvm_test_top
```

```
sv:  uvm_test_top.tb
e:   uvm_test_top.tb.xbus_uvc
e:   uvm_test_top.tb.xbus_uvc.synch
e:   uvm_test_top.tb.xbus_uvc.smp
e:   uvm_test_top.tb.xbus_uvc.bus_monitor
```

Print the short format with types:

```
 % uvm_ml print_tree -type -short
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]
```

## Notes

- The SystemC adapter can not extract type information about the port types because of the native language limitations. Thus, the type name is left empty for SystemC components and ports.

## See Also

- "show units" in Specman Debugging Commands for Use in a UVM-ML Environment

# uvm_ml trace_register_tlm Command

## Purpose

Sets a trace on multi-language TLM registrations.

## Syntax

**uvm_ml trace_register[_tlm] -on** | **-off**

**uvm_ml trace_register[_tlm]**

**uvm_ml trace_register[_tlm] -h** [ **elp** ]

## Parameters

| Parameter | Description |
|---|---|
| **-on** \| **-off** | Turns tracing on or off. |
| <no parameter> | Turns tracing on (implied -on). |
| **-help** | Lists descriptions of all parameters. |

## Description

Automates multi-language TLM registration tracing so that you do not need to write explicit printout messages next to each port/socket registration .

This command lets you see the full hierarchical string names of ports/sockets that need to be connected, potentially to another framework. You use the hierarchical string names as arguments to UVM-ML connection calls.

Thus, this command is especially useful during the integration stage, before writing any connection code. You can turn on the registration tracing before phasing, run the simulation to the beginning of the connect phase, and then observe all the hierarchical names that need to be connected.

## Notes

- This command currently affects UVM-ML SystemVerilog and SystemC only. No tracing is enabled for ***e*** ports.

# Specman Debugging Commands for Use in a UVM-ML Environment

The Specman commands described in this section can help you debug in a UVM-ML environment:

- show units Specman Command
- trace ml_serialization Specman Command
- trace uvm_config Specman Command

## See Also

- "Debugging in Multi-Language Environment" in the *UVM-ML OA User Guide*

## show units Specman Command

### Purpose

Show the ***e*** unit hierarchy.

### Syntax

**sh** [ **o** [ **w** ]] **units**
  [ **-root=** *unit-e-path* ]
  [ **-depth=** *num* | **all** ]
  [ **-with_ports** ]
  [ **-s** [ **hort** [ **_path** [ **s** ]]]]

**sh** [ **o** [ **w** ]] **units -h** [ **elp** ]

## Parameters

| Parameter | Description |
|---|---|
| **-root=** *unit-e-path* | Shows only the specified unit’s subtree. (Default: Show all units.) |
| **-depth=** *num* \| **all** | Depth of the hierarchy to be shown. (Default: 3.)<br>If **-depth=all** is specified, the complete unit tree or subtree is shown. |
| **-with_ports** | Shows both units and ports. (Default: Only units are shown.) |
| **-s hort _path s** | Shows only the appended suffix for each unit in the tree, rather than the full path. |
| **-help** | Prints the description of this command and its options. |

## Description

This command displays the unit tree. The following information is shown for each unit:

- *e-path*
- unique @... *expression* with a hyperlink
- *hdl-path* (if any)
- framework identifier (if there are multiple frameworks in the environment)

If no root unit is specified, the complete unit tree starting from `sys` is shown. In a UVM ML environment with multiple ***e*** trees, all the trees are shown.)

If `-with_ports` is specified, both units and ports in the relevant subtree are shown.

## Example

```
show units -root=sys.my_env -depth=5
```

# trace ml_serialization Specman Command

## Purpose

Trace multi-language serialization operations for objects passed to another framework via TLM interface ports or TLM sockets.

## Syntax

**tra** [ **c** [ **e** ]] **ml_ser** [ **ialization** ]
  **-ports =** *unit.port*
  | **-e-path** *e-path*
  | **-show_commands**
  | **-disable =** *trace-command-index*
  | **-off**

**tra** [ **c** [ **e** ]] **ml_ser** [ **ialization** ] **-h** [ **elp** ]

## Parameters

| Parameter | Description |
|---|---|
| **-ports=** *unit.port* | Specify specific unit types and ports to trace. Wildcards are allowed. |
| ~~**-e**~~-path *e-path* | Specify specific port instances to trace. Wildcards are allowed. |
| **- show_commands** | Displays all previously entered `trace ml_serialization` commands. |
| ~~**-**~~**disable** = *trace-command-index* | Disables the specified `trace ml_serialization` command. You can retrieve the indices for specific commands with the `-show_commands` option. (Wildcards are not allowed.) |
| **-off** | Disables all `trace ml_serialization` commands. |
| **-help** | Prints the description of this command and its options. |

## Description

Turns on tracing of serialization operations in multi-language environments where ports are used to transfer complex data types through language boundaries. Provides details about:

- Serialization of data transferred via interface ports or TLM2 sockets to a foreign language.
- De-serialization of data transferred from a foreign language to interface ports or TLM2 sockets.
- *hdl-path* (if any)
- Type mismatches resulting from de-serialization errors.

Serialization tracing is supported for Specman only at this time.

## Example

The following command turns on multi-language serialization trace for all ports whose ***e*** path starts with “sys.verifier”:

```
cmd-prompt> trace ml_ser -e_path = sys.verifier.*
[13000] client-@3: (started - ML serialization - transport() of 'p1')
           Field: f1 Type: uint Size: 32 bit Value: 4
           Field: f2 Type: list of bit Size: 32 bit Value: {1,0,1,0}
...
[13000] client-@3: (ended - ML serialization - transport() of 'p1')
```

# trace uvm_config Specman Command

## Purpose

Control tracing for `uvm_config_set` and `uvm_config_get` .

## Syntax

**tra c e uvm_config** [ **-off** | **-show** ]

## Parameters

| Parameter | Description |
|---|---|
| **-off** | Turn `uvm_config` tracing off. |
| **-show** | Show whether `uvm_config` tracing is off or on. |

## Description

In ***e*** , both `uvm_config_set` and `uvm_config_get` are declared as generation constraints and executed during the generation phase. When `trace uvm_config` is on:

- All invocations of a keep uvm_config_set() constraint, each of which adds an entry to the configuration database, are reported.
- All invocations of a keep uvm_config_get() constraint, each of which queries the configuration database for the value of a given field, are reported.
  This happens regardless of whether the retrieved value is actually used. For example, if the constraint is soft and is overridden by another constraint, it is still reported by the trace.
- Within UVM ML environments, uvm_config settings done in a language other than e are also reported by the trace.

## trace uvm_config_set Messages

The following information is included in the trace messages for `uvm_config_set` :

- The ***e*** path of the unit (or the corresponding full name of a foreign component if the setting is done in a different language) in the context of which the setting is executed.
- The relative path name and the field name (both may include wild cards) used in the setting.
- The value used in the setting.

⚠ If the setting is done outside e and the value type is a scalar type or a packed struct type, Specman gets the value as a bit stream, and the actual type is unknown. In this case, the value is shown in hexadecimal numeric form. Non-packed structs and strings are reported as is.

- The source line and module where the constraint appears (and a message if the setting is done outside e).

## trace uvm_config_get Messages

The following information is included in the trace messages for `uvm_config_get` :

- The e path of the unit containing the field that is queried.
- The name of the field that is queried.
- The source line and module where the constraint appears.
- The field value (or a message that no value is found).

## Keeping the State across reload/restore

By default, the `trace uvm_config` state, whether on or off, is retained across reload/restore.

## uvm_config_set Examples

1. A constraint like the following:

   ```
   keep uvm_config_set("my_agent.*", "data", 157);
   ```

   Produces a trace message like the following:

   ```
   [trace uvm_config] my_env-@10 sys.my_env: uvm_config_set("my_agent.*", "data",
   157) done at line 20 in @my_env
   ```

2. A constraint like the following:

   ```
   keep uvm_config_set("*", "env_name", me.name );
   ```

   Produces a trace message like the following (if the value of me.name is “foo”):

   ```
   [trace uvm_config] my_env-@10 sys.my_env: uvm_config_set("*", "env_name", "foo")
   done at line 25 in @my_env
   ```

3. A constraint like the following:

keep uvm_config_set("my_agent", "p", me.packet);

Produces a trace message like the following:

```
[trace uvm_config] my_env-@10 sys.my_env: uvm_config_set("my_agent", "p",
packet_s-@18) done at line 30 in @my_env
```

4. Settings done outside ***e*** produce messages like the following:

```
[trace uvm_config] topenv.inst1: uvm_config_set("u1", "name", "bar") done outside
Specman
[trace uvm_config] topenv.inst1: uvm_config_set("u1", "data", 0x73DF) done outside
Specman
```

   The message above indicates that Specman received the value as a bit stream.

## uvm_config_get Examples

1. A constraint like the following:

```
keep uvm_config_get(data);
```

   Produces a trace message like the following:

```
[trace uvm_config] agent-@15 tovenv.inst1.my_e_agent: uvm_config_get(data) done at
line 10 in @my_agent, value is 1375
```

2. A constraint like the following:

```
keep uvm_config_get(addr);
```

   Produces a trace message like the following if no value is found:

```
[trace uvm_config] agent-@15 tovenv.inst1.my_e_agent: uvm_config_get(addr) done at
line 12 in @my_agent, no value found
```

## See Also

- UVM-ML e Configuration

# IES Simulator reset Command

The IES simulator `reset` command resets the simulation time to time zero, while generally preserving the IES debugging environment. The use of the IES simulator `reset` command is supported for UVM-ML adapters, with the caveat that you follow the usage notes described in this section.

## Usage Notes for IES Simulator Reset

- For all C++ user code (that is not SystemC code), you must:
    - Either compile the code with the `-Bsymbolic` linker switch (or the comparable switch for non-GNU compilers)
    - Or unload the code during the simulator reset
- IES simulator reset is not supported when ***e*** testbench communicates with the SystemC DUT / testbench directly and DPI libraries are present in the testbench at the same time.
- The Specman `configure ies -sync_reset` command is supported with the following limitations:
    - The `pre-test` reset option restores the Specman state to what it was prior to the simulator `run` command, instead of to what it was prior to the `sn test` command. (The `sn test` command does not exist in UVM-ML.)
    - The `pre-run` reset option is not supported for UVM-ML. You get an error message if you configure Specman to work in this mode in a UVM-ML environment.

⚠ The IES simulator `reset` command is supported for UVM-ML starting with the following IES releases: 13.20.032, 14.10.021, and 14.20.006