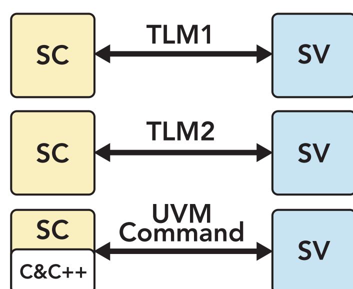
# Introducing UVM Connect

*by Adam Erickson, Verification Technologist, Mentor Graphics*

This article introduces Mentor's new open-source UVM Connect (UVMC) library, now available for download at verificationacademy.com. UVMC provides TLM1 and TLM2 connectivity and object passing between SystemC and SystemVerilog models and components. It also provides a UVM Command API for accessing and controlling UVM simulation from SystemC (or C or C++).




## ENABLING IP & VIP REUSE

So what does this new capability allow you to do? UVM Connect enables the following use models, all designed to maximize IP reuse:

***Abstraction Refinement—*** Reuse your SystemC architectural models as reference models in UVM verification. Reuse your stimulus generation agents in SystemVerilog to verify models in SystemC.

***Expansion of VIP Inventory***—More off-the-shelf VIP is available when you are no longer confined to VIP written in one language. Increase IP reuse! To properly verify large SoC systems, verification environments are becoming more of an integration problem than a design problem.

***Leveraging language strengths***—Each language has its strengths. You can leverage SV's powerful constraint solvers and UVM's sequences to provide random stimulus to your SC architectural models. You can leverage SC's speed and capacity for verification of untimed or loosely timed system-level environments.

***Access to SV UVM from SC***—The UVM Command API provides a bridge between SC and UVM simulation in SV. With this API you can wait for and control UVM phase transitions, set and get configuration, issue UVM-style reports, set factory type and instance overrides, and more.

## KEY FEATURES

The UVM Connect library makes connecting TLM models in SystemC and UVM in SystemVerilog a relatively straightforward process. This section enumerates some keys features of UVM Connect.

***Simplicity***—Object-based data transfer is accomplished with very little preparation needed by the user.

***Optional***—The UVMC library is provided as a separate, optional package to UVM. You do not need to import the package if your environments do not require cross-language TLM connections or access to the UVM Command API.

***Works with Standard UVM***—UVMC works out-of-box with open-source Accellera UVM 1.1a and later. UVMC can also work with previous UVM open-source releases with one minor change.

***Encourages native modeling methodology***—UVMC does not impose a foreign methodology nor require your models or transactions to inherit from a base class. Your TLM models can fully exploit the features of the language in which they are written.

***Supports existing models***—Your existing TLM models in both SystemVerilog and SystemC can be reused in a mixed-language context without modification.

***Reinforces TLM modeling standards***—UVMC reinforces the principles and purpose of the TLM interface standard-designing independent models that communicate without directly referring to each other. Such models become highly reusable. They can be integrated in both native and mixed-language environments without modification.

## MAKING UVMC CONNECTIONS

To communicate, verification components must agree on the data they are exchanging and the interface used to exchange that data. TLM connections parameterized to the type of object (transaction) help components meet these requirements and thus ease integration costs and increase their reuse. To make such connections across the SC-SV language boundary, UVMC provides *connect* and *connect_hier* functions.

***SV TLM2:***

uvmc_tlm #(trans)::**connect** (port_handle, “lookup”);
uvmc_tlm #(trans)::**connect_hier** (port_handle, “lookup”);

***SC TLM2:***

**uvmc_connect** (port_ref, “lookup”);
**uvmc_connect_hier** (port_ref, “lookup”);

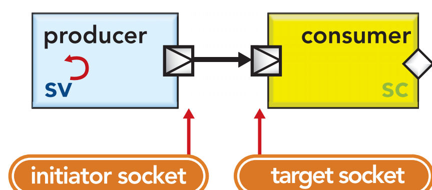
***trans***—Specifies the transaction type for unidirectional TLM 1 ports, export, and imps. Needed only for SV.

***port_handle or ref***—The handle or reference to the port, export, imp, interface, or socket instance to be connected.

***lookup***—An optional lookup string to register for this port.

UVMC registers the port’s hierarchical name and lookup string (if provided) for later matching against other port registrations within both SV and SC.  A string match between any two registered ports results in those ports being connected—whether the components are in the same or different languages.

Let’s see how we use the connect function in a real example. The diagram below shows an SV producer component and an SC consumer component communicating via a TLM socket connection.




The following code creates the testbench and UVMC connection in both SV and SC. *This code is complete and executable.* It is all you need to pass *tlm_generic_payload* transactions between a SystemC and SystemVerilog component.

***SystemVerilog:***

```
import uvm_pkg::*;
import uvmc_pkg::*;
`include “producer.sv”

module sv_main;
  producer prod = new(“prod”);
  initial begin
    uvmc_tlm #()::connect(prod.out, “foo”);
    run_test();
  end
endmodule
```

***SystemC:***

```
#include “uvmc.h”
using namespace uvmc;
#include “consumer.h”

int sc_main(int argc, char* argv[]) {
  consumer cons(“cons”);
  uvmc_connect(cons.in,”foo”);
  sc_start(-1);
  return 0;
}
```

The *sv_main* top-level module creates  the SV portion of the example. It creates an instance of a producer component, then registers the producer’s *out* initiator socket with UVMC using the lookup string “foo”. It then starts UVM test flow with the call to *run_test()*.

The *sc_main* function creates the SC portion of this example. It creates an instance of a consumer sc_module, then registers the consumer's *in* target socket with UVMC using the lookup string, "foo". It then starts SC simulation with the call to *sc_start*.

During elaboration, UVMC will connect the producer and consumer sockets because they were registered with the same lookup string.

(Note: SV-side TLM port connections would normally be made in UVM's *connect_phase*. We omit this for the sake of brevity.)

## TRANSACTION CONVERSION

Object transfer requires converters to translate between the two types when crossing the SC-SV language boundary.

*UVMC provides built-in support for the TLM generic payload (TLM GP). You don't need to do anything regarding transaction conversion when using TLM GP.*

Use of the TLM GP also affords the best opportunity for interoperability between independently designed components from multiple IP suppliers. Thus, there is strong incentive to use the generic payload if possible.

If, however, you are not using the TLM GP, the task of creating a converter in SC and SV is fairly simple.

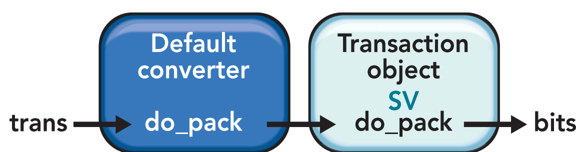
Let's say we have a transaction with a command, address, and variable-length data members. The basic definitions in SV and SC might look like this:

| SV | SC |
|---|---|
| class C; | class C { |
| cmd_t cmd; | cmd_t cmd; |
| int unsigned addr; | unsigned int addr; |
| byte data[$] | vector<char> data; |
| ... | }; |
| endclass | |

UVMC uses separate converter classes to pack and unpack transactions. This allows you to define converters independently from the transactions they operate on. UVMC defines default converter implementations that conform to the prevailing methodology in each language. Yet, UVMC can accommodate almost any application, such as using a SV-side object that does not extend *uvm_object*.

### *SV-side conversion*

In SV, by default, UVMC default converter delegates conversion to the transaction's *pack* and *unpack* methods. Having the transaction class itself implement its own conversion is the *UVM way*, so UVMC accommodates.




UVM transaction conversion is usually implemented in its *do_pack* and *do_unpack* methods, or via the *`uvm_field* macros. The *do_pack/unpack* method implementations have better performance and provide greater flexibility, debug, and type-support than the *`uvm_field* macros. The extra time to implement these methods is far outweighed by the cumulative performance and debug benefits they afford.

The recommended way to implement *do_pack/unpack* is to use a set of `uvm_pack `uvm_unpack macros, which are part of the UVM standard. These macros are of the "good" variety. They are small and more efficient than the *`uvm_field* macros and *uvm_packer* API.

The following transaction definition implements *do_pack/unpack* in the recommended way:

```
class packet extends uvm_sequence_item;

  `uvm_object_utils(packet)

  rand cmd_t cmd;
  rand int   unsigned addr;
  rand byte  data[$];

  constraint C_data_size {
    data.size inside {[1:16]};
  }
```

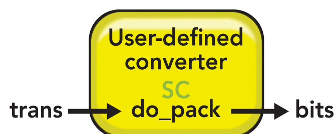
```
function new(string name="");
    super.new(name);
  endfunction

  function void do_pack(uvm_packer packer);
    super. do_pack(packer);
    `uvm_pack_int(cmd)
    `uvm_pack_int(addr)
    `uvm_pack_queue(data)
  endfunction
```

```
  function void do_unpack(uvm_packer packer);
    super. do_unpack(packer);
    `uvm_unpack_int(cmd)
    `uvm_unpack_int(addr)
    `uvm_unpack_queue(data)
  endfunction
  ...
endclass
```

### *SC-side conversion*

Transaction classes in SystemC do not typically extend from a common base class, so conversion is performed in a separate converter class.




An SC-side converter is actually a template specialization of the default converter, *uvmc_converter<T>*. Template specializations allow you to override the generalized implementation of a parameterized class with one that is custom-tailored for a specific set of parameter values, e.g. *uvmc_converter<packet>*. This is a compiler-level operation, so any code that uses *uvmc_converter<packet>* automatically gets our specialized implementation.

The following code implements a converter class for our packet transaction class:

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

template <>
class uvmc_converter<packet> {
  public:
  virtual void do_pack(const packet &t,
                       uvmc_packer &packer) {
    packer << t.cmd << t.addr << t.data;
  }
  virtual void do_unpack(packet &t,
                         uvmc_packer &packer) {
    packer >> t.cmd >> t.addr >> t.data;
  }
};
```

The packer variable is an instance of *uvmc_packer*, the counterpart to *UVM's uvm_packer* on the SV side. You can stream your transaction members are streamed to and from a built-in packer instance variable much like you would stream them to standard out using cout. Use *operator*<< for packing, and *operator*>> for unpacking.

Except for the actual field names being streamed, most converter classes conform to this template structure. So, as a convenience, UVMC provides macros that can generate a complete *uvmc_converter*<*T*> class <u>and</u> output streaming capability for your transaction with a single *UVMC_UTILS* macro invocation. Unlike the `uvm_field macros in UVM, these macros expand into succinct, human-readable code (actually, the exact code shown above).

Using the macro option, our custom converter class definition for *packet* reduces to one line:

```
#include "uvmc.h"
using namespace uvmc;
UVMC_UTILS_3 (packet, cmd, addr, data)
```

The number suffix in the macro name indicates the number of transaction fields being converted. Then, simply list each field member in the order you want them streamed. Just make sure the order is the same as the packing and unpacking that happens on the SV side. UVMC supports up to 20 fields.

### *Type Support*

For streaming fields in your converter implementations, UVMC supports virtually all the built-in data types in SV and SC. In SV, all the integral types, reals, and single-dimensional, dynamic, queue, and associative arrays of any of the built-in types have direct support. In SC, all the integral types and float/double are support, as are fixed arrays, vectors, lists, and maps of these types. The integral SystemC types such as sc_bv<N> and sc_uint<N> are also supported. Rest assured, any type that does not have direct support can be adapted to one of the supported types.

### *On (not) using `uvm_field macros*

The *`uvm_field* macros hurt run-time performance, can make debug more difficult, and can not accomodate custom behaviors (e.g. conditional packing, copying, etc. based on value of another field).

The macros generate far more code than is necessary to implement the operations directly. This consumes memory and hurts performance. Even a small performance decrease of 1% can dramatically affect regression times and lowers the ceiling on potential accelerator/emulator performance gains.

### *Transaction requirements*

UVM Connect imposes very few requirements on the transaction types being conveyed between TLM models in SV and SC, a critical requirement for enabling reuse of existing IP. The more restrictions you impose on the transaction type, the more difficult it will be to reuse models that use those transactions.

- ***No base classes required.*** It is not required that the transaction inherit from any base class--in either SV or SC--to facilitate their conversion
- ***No factory registration required.*** It is not required that the transaction register with a factory—be it the UVM factory or any user-defined factory mechanism.
- ***No converter API required.*** It is not required that the transaction provide the conversion methods. You can specify a different converter for each or every UVMC connection. You can define multiple converters for the same transaction type.
- ***Transaction equivalence not required.*** It is not required that the members (properties) of the transaction classes in each language be of equal number, equivalent type, and declaration order. The converter can adapt to disparate transaction definitions at the same time it serializes the data.

## UVM COMMAND API

The UVM Command API gives SystemC users access to key UVM features in SystemVerilog. These include:

- Wait for a UVM to reach a given simulation phase
- Raise and drop objections to phases, effectively controlling UVM test flow
- Set and get UVM configuration, including objects
- Send UVM report messages, and set report verbosity
- Set type and instance overrides in the UVM factory
- Print UVM component topology

To enable use of the UVMC command API, you must call *uvmc_init()* from an initial block on the SystemVerilog side. This function starts a background process that services UVM command requests from SystemC.

```
module sv_main;

  import uvm_pkg::*;
  import uvmc_pkg::*;

  initial begin
    uvmc_cmd_init();
    run_test();
  end

endmodule
```

SC-side calls to the UVM Command API will block until SystemVerilog has finished elaboration and the *uvmc_init()* function has been called. For this reason, such calls must be made from within SC thread processes.

### *Issuing UVM reports from SystemC*

UVMC provides an API into UVM's reporting mechanism, allowing you to send reports to UVM's report server and to set report verbosity at any context of the UVM hierarchy. As with natively issued UVM reports, all reports you send to UVM are subject to filtration by configured verbosity level, actions, and report catchers.

Just as in UVM, UVMC provides convenient macro definitions for sending reports efficiently and with automatic SystemC-side filename and line number information.

```
UVMC_INFO("SC-TOP/SET_CFG",
     "Setting config for SV-side producer",
      UVM_MEDIUM,"");
```

### *Set and Get Config*

UVMC supports setting and getting integral, string, and object values from UVM's configuration database.

Use of configuration objects is strongly recommended over one-at-a-time integrals and strings. You can pass configuration for an entire component or set of components with a single call, and the configuration object is type-safe

to the components that accept those objects. With ints and strings, it's far too easy to get configuration wrong (which can be hard to debug), especially with generic field names like "count" and "addr".

Before you can pass configuration objects, you will need to define an equivalent transaction type and converter on the SystemC side. As shown earlier, this is easily done:

```
#include "uvmc.h"
#include "uvmc_macros.h"
using namespace uvmc;

class prod_cfg {
  public:
  int min_addr, max_addr;
  int min_data_len, max_data_len;
  int min_trans, max_trans;
};

UVMC_UTILS_6(prod_cfg, min_addr, max_addr,
            min_data_len, max_data_len,
            min_trans, max_trans)
```

We use the above definition to configure a SV-side stimulus generator and a couple of "loose" int and string fields. Note the form is much the same as in UVM.

```
uvmc_set_config_int ("e.agent.driver","",
                                  "max_error", 10);

uvmc_set_config_string ("e.agent.seqr",
                                    "run_phase",
                                    "default_sequence",
                                    s.c_str());

prod_cfg cfg = new();
cfg.min_addr = 'h0100; cfg.max_addr = 'h0FFF;
cfg.max_data_len = 10; cfg.max_trans = 100;

uvmc_set_config_object ("prod_cfg", "e.prod",
                                     "", "config", cfg);

if (!uvmc_get_config_int ("sc_top", "dut",
                     "max_error", max_error))
  UVMC_ERROR("NO_MAX_SET",
                 "max_error not set",name());
```

### *Phase Control*

With UVMC, you can wait for a UVM phase to reach a given state, and you can raise and drop objections to any phase, thus controlling UVM test flow.

```
uvmc_wait_for_phase("run",UVM_PHASE_STARTED);

uvmc_raise_objection("run",
                                  "SC producer active");
  // produce data

uvmc_drop_objection("run",
                                 "SC producer done");
```

### *Factory*

You can set UVM factory type and instance overrides as well. Let's say you are testing a SC-side slave device and want to drive it with a subtype of the producer that would normally be used. Once you make all your overrides, you can check that they "took" using some factory debug commands.

```
uvmc_set_factory_type_override("producer",
                              "producer_ext","e.*");
  uvmc_set_factory_inst_override("scoreboard",
                              "scoreboard_ext","e.*");

// factory should show overrides
uvmc_print_factory();

// show information about how the factory
// chooses which type to create. Should be
// the "_ext" versions in this case.

uvmc_debug_factory_create("producer",
                              "e.prod");
uvmc_debug_factory_create("scoreboard",
                              "e.sb");

// get the type the factory would produce
// give a requested type and context. We
// can use the result in subsequent overrides
string override = uvmc_find_factory_
                    override("producer","e.prod");
```

### Printing UVM Topology

You can print UVM testbench topology from SystemC. Just be sure you invoke the command *after* UVM has built the testbench.

```
uvmc_wait_for_phase("build",UVM_PHASE_ENDED);

cout << "UVM Topology:" << endl;

uvmc_print_topology();
```

### Parting Thoughts

UVM Connect bridges the SystemC and SystemVerilog language boundary to provide seamless TLM1 and TLM2 connectivity between components residing in those two languages. It also provides a means of directly accessing and controlling UVM simulation via the UVM Command API.

The library is available for download on the verificationacademy.com website. It is distributed under the Apache license, meaning it is free for you to use, modify, and share. While the library was written to be portable across all three simulators (it uses standard SystemVerilog and SystemC), Mentor could not qualify the product for use on non-Questa simulators. It is expected that a small handful of issues may need addressing when compiling on non-Questa simulators. We will happily accept any suggested modifications that would allow UVM Connect to be qualified on other simulators.

## REFERENCES & RESOURCES

A partial list of sources for information on SystemC, SystemVerilog, UVM, and related topics

Standard SystemC Language Reference Manual; IEEE 1666-2011, March 8, 2011. http://standards.ieee.org/getieee/1666/download/1666-2011.pdf

IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language; IEEE 1800-2009; December 11, 2009

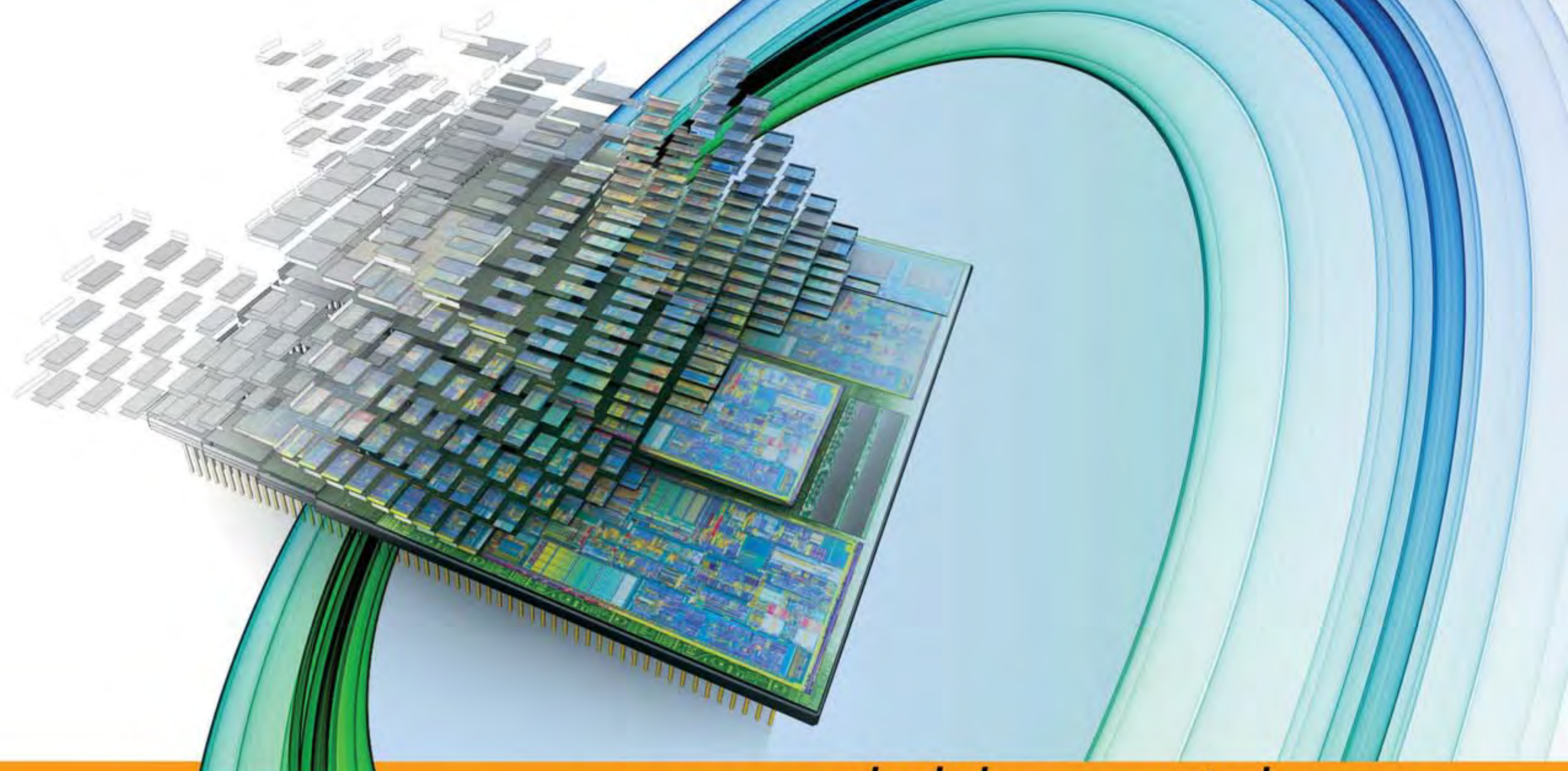
OSCI TLM 2.0 Language Reference Manual, version TLM 2.0.1, Document JA32; copyright Open SystemC Initiative, July 2009 (absorbed into [1] above)

Universal Verification Methodology (UVM) Accellera; http://www.accellera.org/activities/vip

Verification Academy - http://www.verificationacademy.com/UVMConnect

UVM Cookbook - http://verificationacademy.com/uvm-ovm

Are UVM/OVM Macros Evil? A Cost-Benefit Analysis\u201d; DVCon 2011, Feb 2011. http://verificationacademy.com/uvm-ovm/MacroCostBenefit



# verification HORIZONS

Editor: Tom Fitzpatrick
Program Manager: Rebecca Granquist

Wilsonville Worldwide Headquarters
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777
Phone: 503-685-7000

To subscribe visit:
www.mentor.com/horizons

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

Mentor Graphics®

www.mentor.com