# Transaction-Level Friending:

## An Open-Source, Standards-Based Library for Connecting TLM Models in SystemC and SystemVerilog

Adam Erickson
Verification Methodologist
Mentor Graphics Corporation
Adam_Erickson@mentor.com

**This paper describes the purpose, requirements, development challenges, and applications of an open-source library for establishing standard TLM-based communication between SystemC (SC) and SystemVerilog (SV) models, including C/C++ models wrapped in SC or SV. It also describes a SystemC-side interface for controlling simulations based on the Universal Verification Methodology (UVM) in SystemVerilog. The UVM Connect library is available for download and has been proven to work on three major EDA vendors' simulators [1].**

### I. INTRODUCTION

A 2010 Wilson Research Group independent study of trends in functional verification confirmed that usage of SystemVerilog, SystemC, C, and C++ was growing, as was deployment of the Universal Verification Methodology (UVM), a SystemVerilog library of testbench building blocks and best practices. [2]

Each of the language and library standards used by the verification community possesses strengths suited to its intended purpose:

- C and C++ for software development and untimed modeling [3] [4]
- SystemC for high-speed architectural modeling of hardware and early collaboration with software teams [5]
- TLM 2.0 for interoperable, approximately timed (AT) and loosely timed (LT) transaction-level communication between independently design models [5]
- SystemVerilog for RTL and gate-level simulation, behavioral testbenches, constrained random stimulus, functional coverage, and assertions [6]
- UVM for development of modular, reusable, scalable testbenches; sequence-based stimulus, and verification methodology [7] [8]

With all these languages and other standards at play, it should be no surprise that use of hybrid or multi-language SoC testbenches is growing too. The 2011 merger between the Open SystemC Initiative (OSCI), the body responsible for the SystemC and TLM 1.0/2.0 standards, and Accellera, the body responsible for standardizing the UVM and initiating the standardization of SystemVerilog, could be viewed as a reflection of this trend. The merger was explained thus:

> *System, software and semiconductor design activities are converging to meet the increasing challenges to create complex system-on-chips (SoCs).*
> ...
> *The relationship between OSCI's TLM-2.0 SystemC Transaction Level Modeling standard and Accellera's Universal Verification Methodology (UVM) standard exemplifies the synergy that exists between the two organizations. [9]*

A direct result of this synergy is UVM Connect, which integrates all these standards to enable connections between TLM models written in SystemC and SystemVerilog. It also includes an API that allows SystemC, C, and C++ code to interact with and control the execution of UVM testbenches in SystemVerilog.

### II. REQUIREMENTS

Establishing a communications link between SystemC and SystemVerilog is nothing new. There have been plenty of implementations, papers, and proposals written describing how to interface SystemVerilog with a foreign language [10] [11] [12] [13] [14].

We believe UVM Connect differentiates itself by virtue of the unique requirements that drove its development, including:

- Open-source, to foster broad industry acceptance and transparency.
- Portability across vendors, to protect user investment and allow vendors to differentiate with layered products.
- Must work without requiring modifications to existing standards, Accellera's UVM in particular.
- Allow models to fully exploit the features of the language in which they were written.
- Does not impose a foreign methodology nor require models or transactions to inherit from a base class, implement a conversion interface, etc.
- Supports the connection of existing TLM models in both SystemC and SystemVerilog without requiring their modification.

Having met all these requirements, we believe UVM Connect provides a natural solution to the mixed-language feature being discussed by the Accellera VIP Technical Steering Committee (TSC).

## III. APPLICATIONS

This section describes some of the new and interesting use models that are possible when you can combine software, architectural modeling, RTL verification, and emulation activities.

Above all, we wanted UVM Connect to make inter-language communication easy, to allow using the language that best fits the application. We wanted to increase available SC, C, and C++ IP in UVM testbenches, to increase available VIP in SC, C, and C++ verification—including all existing IP and VIP. For example, SystemVerilog provides powerful constraint solver and coverage collection capabilities, and there are many SystemVerilog-based, TLM-ready VIP to choose from. Why not use these to verify SC models?

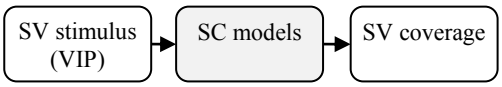


Figure 1. Leveraging SV strengths in SC model verification

The following are just a sampling of use models enabled with UVM Connect.

### A. Use SystemVerilog testbench to verify SystemC, C, and C++ models

To verify a SystemC model (or a C/C++ model that you've wrapped with a thin layer of SC), you can leverage UVM best practices and SystemVerilog features to develop high-level test stimulus, capture functional coverage, etc.

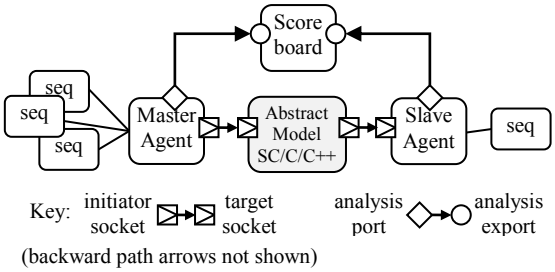


Figure 2. UVM SV testbench verifying SC/C/C++ model

If later you have an RTL representation of the abstract model, you can reuse the UVM testbench and test suite to kick start RTL verification. Because the same tests and constraint solver are being used, even randomized stimulus is reproducible. Of course, you would still need to augment the high-level tests with RTL-level tests to verify timing and interactions with other RTL models.

### B. Use SystemC, C, and C++ models in RTL verification

SC models (and C/C++ models that you've wrapped with a thin layer of SC) can be more easily integrated as reference models in your UVM testbench.

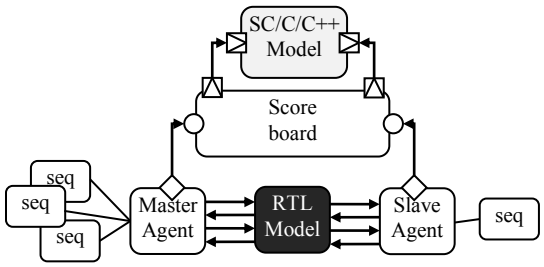


Figure 3. Us

### C. Virtual Prototyping

Some SV VIP can operate in high-speed transactional mode. These VIP (and non-TLM VIP that you've adapted to work in TLM mode) can be used in a SystemC/C/C++ SoC testbench for early software testing, memory map and interconnect verification, and other high-level activities.

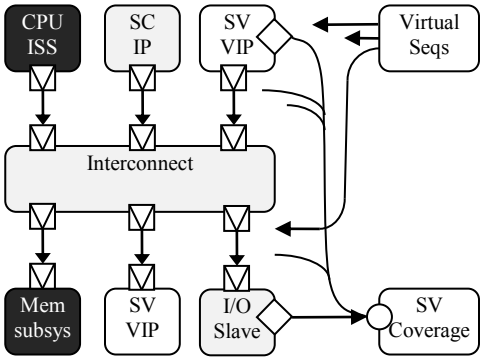


Figure 4. SV → SC

### D. Foreign model import

Sometimes your verification teams are versed in one language and would prefer to work exclusively in a single language. UVM Connect allows you to implement a local *proxy* model that delegates to a connected model in another language. The proxy model registers cross-language connections to the foreign model in its constructor, thus hiding the cross-language connection details to external components (and users). You can then integrate the proxy as an ordinary, native component. The resulting testbench appears to the user to be a homogenous, single-language environment.

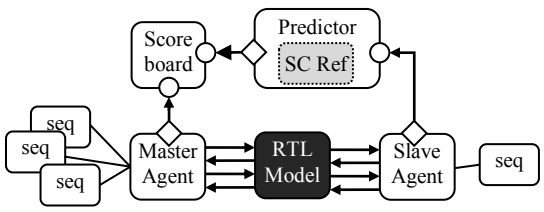


Figure 5. SV → SC

### E. RTL integrated into TLM platform

In this application, you integrate RTL models in your abstract SoC testbench using a component that adapts between TLM 2.0 sockets and the RTL interface. Some VIP

can be configured to do this for you. This enables you to continually validate the hardware software interface as RTL becomes available. The same test infrastructure is used throughout the process of refinement from early software development through RTL verification.

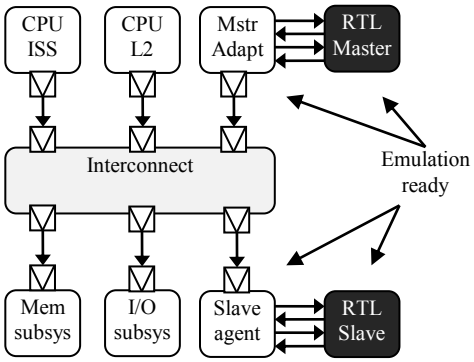


Figure 6. SV → SC

### F. Other Applications

UVM Connect has been used and is being evaluated for use in several other applications, including:

- Establishing a TLM link between simulator and emulator
- Being evaluated to facilitate coordination of embedded software and the UVM portions of an SoC testbench.

Of course, these are not the only applications possible when C, C++, SC, and SV can coexist with relative ease. In each case, TLM models are written in the language best suited for their initial purpose. When appropriate, such models are then reused in different applications, including those involving another language.

UVM Connect is implemented as a SystemVerilog package and a SystemC namespace. They contain functions for registering TLM models' ports and sockets for cross-language connection, conversion of transactions between languages in support of those connections, and enabling SystemC to interact with and control UVM testbenches.

First, we look at making TLM connections.

## IV. Making Connections

The TLM 2.0 standard defines an API for passing transactions between models using initiator and target sockets. A model's initiator socket is bound to another model's target socket and transaction (objects) are transferred between them. UVM Connect allows these models to reside in different languages.

To accomplish this, UVM Connect provides a *connect* function in both languages that serves to register a socket, port, or export for cross-language connection:

```
SystemVerilog TLM2:
uvmc_tlm #(T)::connect(socket_h, "lookup");
SystemC TLM2:
uvmc_connect( socket_ref, "lookup");
```

Language differences affect the syntax of the *connect* call on each side, but the semantic is the same.[1] Each *connect* call registers the socket, port, or export given in the 1st argument with a lookup string given in the 2nd argument. During elaboration, the lookup strings are used to determine connection pairs. Port directions and interface types are checked to help ensure the connections are compatible.

Example:

Our first example shows how to use these *connect* functions to connect a SystemVerilog producer model's initiator socket to a SystemC consumer's target socket. Here, the producer might be a stimulus generator (agent), and the consumer might be a SystemC architectural model.

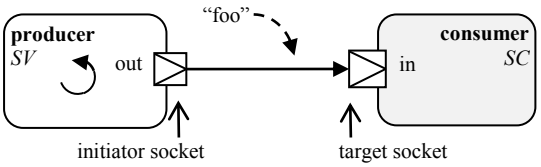


Figure 7. SV → SC

The SystemVerilog and SystemC code needed to create these two components and establish their connection is as follows: [2]

```
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();
  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()*.

[1] UVM Connect methods and classes are prefixed with uvmc_.

[2] For brevity, the example UVM code will invoke new() directly instead of using the UVM factory.

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 two sockets because they were registered with the same lookup string, "foo". We did not specify a transaction type in the connect call, so the TLM 2.0 generic payload is implied.

Except for the producer and consumer models themselves, the above code is complete. This is all you need to establish a cross-language connection between TLM components using the generic payload transaction type.

The next example adds an SV scoreboard whose *expect* export is connected natively to the producer's analysis port and whose *actual* analysis export is connected to the SC consumer.

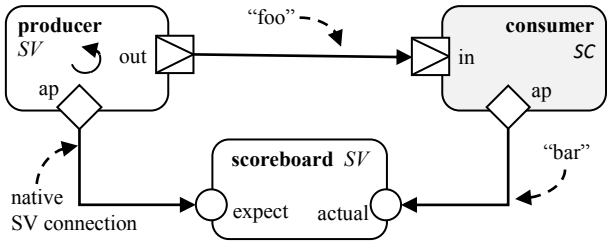


Figure 8. SV → SC → SV

SystemVerilog:

```
module sv_main;
  producer prod = new("prod");
  scoreboard sb = new("sb");
  initial begin
    prod.ap.connect(sb.expect); // native conn.
    uvmc_tlm  #(uvm_tlm_gp)::connect(prod.out,
                                     "foo");
    uvmc_tlm1 #(uvm_tlm_gp)::connect(sb.actual,
                                     "bar");
    run_test();
  end
endmodule
```

SystemC:

```
int sc_main(int argc,char* argv[]) {
  consumer cons("consumer");
  uvmc_connect(cons.in,"foo");
  uvmc_connect(cons.ap,"bar");
  sc_start();
  return 0;
}
```

In this example, we establish another cross language connection using a different lookup string, "bar". Analysis ports are part of the TLM1 standard, we use a slightly different calling syntax when making this connection. TLM2 connections in SV are made using the *uvmc_tlm* class, while TLM1 connections in SV use the *uvmc_tlm1* class. Function template overloading in C++ helps us retain the same syntax for both TLM1 and TLM2 connections.

UVM Connect will produce an error if the same lookup string is used to register more than one cross-language connection In practice, most testbenches have few cross-language connections, so ensuring each connection has a unique lookup string is typically not a problem. For large testbenches with many connections, centralizing the lookup strings in a common side file might help mitigate the problem. It is *not* recommended that the lookup string use a port or socket's hierarchical name, i.e. using *get_full_name*(), because the other side would have to use this context string to make a match. When (not if) the socket changes hierarchical location (as during block to system reuse), you would have to modify the connect calls on the other side.

The next example shows how to handle stimulus coming from the SC side. In UVM SV, time-consuming activity occurs during the *run_phase*. The *run_phase* will end when there are no components that object to it ending. If an SC model wants to participate in time-consuming activity during UVM's *run_phase*, it must raise an objection to UVM ending that phase and hold that objection until it has finished with its activity.

The following example demonstrates an SC model driving an SV consumer (say, an RTL bus agent). To avoid modifying the existing producer, we use inheritance to define a new producer that can control UVM's *run_phase*.

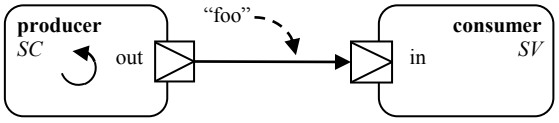


Figure 9. SC → SV

The code below extends the SC producer to have a background process, *objector*. This process will use the UVM command API to raise an objection to the *run_phase*, even before the *run_phase* begins. The process then waits until the base producer has completed its stimulus, which is indicated by a *done* event. The objection to the *run_phase* is then dropped, which allows the UVM test flow to continue onto the next phase.

SystemC:

```
struct prod_uvm : public producer {
  prod_uvm(sc_module_name nm) :
           producer(nm) {
    SC_THREAD(objector);
  }
  SC_HAS_PROCESS(prod_uvm)
  void objector() {
    uvmc_raise_objection("run");
    wait(done);
    uvmc_drop_objection("run");
  }
};
```

Now, all we need to do is use the UVM-aware producer model on the SC side, making the TLM connections as we did previous examples:

SystemC:
```
int sc_main(int argc,char* argv[]) {
  prod_uvm prod("producer"); // UVM-aware version
  uvmc_connect(prod.out,"42");
  sc_start();
  return 0;
}
```
SystemVerilog:
```
module sv_main;
 consumer cons = new("cons");
 initial begin
  uvmc_tlm #()::connect(cons.in,"42");
  uvmc_init(); // init cmd API
  run_test();
 end
endmodule
```

Note that, on the SV side, we added a call to *uvmc_init*. This function starts up a background process that services incoming UVM command requests.

Our final example demonstrates how you can leverage UVM Connect hierarchical connection capability to wrap foreign models in a native skin. Doing so allows you to present to users what appears as a pure native environment. Here, we use the *uvmc_connect_hier* function to register hierarchical connections. Except for the *_hier* suffix, the syntax of *uvmc_connect_hier* is the same as with *connect*.

First, we define a SC producer, but provide no implementation. Instead, we use uvmc_connect_hier to connect a SV-side producer implementation to our initiator socket.

SystemC:
```
class producer: public sc_module {
  public:
  tlm_initiator_socket<32> out;
  SC_CTOR(producer) : out("out") {
    uvmc_connect_hier(out,"1234");
  }
};
```

The SC producer's initiator socket will be driven by the SV producer, which is instantiated independently in SV as in previous examples:

SystemVerilog:
```
module sv_main;
 producer prod = new("prod");
 initial begin
  uvmc_tlm #()::connect(prod.out,"1234");
  run_test();
 end
endmodule
```

With the above code compiled in, you can then define your environments using only native, SC models:

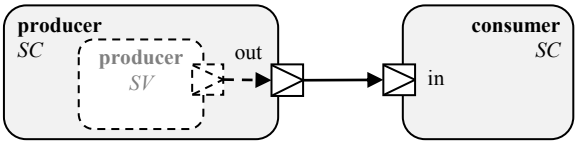


Figure 10. Transparent embedding of SV models in SC testbench

SystemC:
```
#include "producer.h"
#include "consumer.h"
int sc_main(int argc, char* argv[])
{
  producer prod("producer");
  consumer cons("consumer");
  prod.out.bind(cons.in);
  sc_start();
  return 0;
};
```

## V. Transaction Conversion

To pass transactions between the two languages, UVM Connect converts the content of each transaction into a bit stream that is passed via SystemVerilog's DPI-C interface. The previous examples demonstrated built-in support for the TLM 2.0 Generic Payload transaction type, which makes the conversion process transparent for the majority of TLM model developers.

When transaction types other than the generic payload are needed, conversion routines must be provided. You can define a single conversion algorithm for all connections of a given transaction type, or you can specify a different conversion algorithm for each connection you make.

### A. SystemVerilog Conversion Options

For SystemVerilog UVM transactions, the standard way to define a conversion algorithm is to implement the transaction's *do_pack* and *do_unpack* virtual methods. We recommend using the standard *`uvm_pack* and *`uvm_unpack* convenience macros to do this:

SystemVerilog:
```
class packet extends uvm_sequence_item;

  `uvm_object_utils(packet)

  rand cmd_t cmd;
  rand int unsigned addr;
  rand byte data[$];
  ...
  function void do_pack(uvm_packer packer);
    `uvm_pack_int(cmd)
    `uvm_pack_int(addr)
    `uvm_pack_queue(data)
  endfunction

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

Another alternative is to implement a UVM transaction's *pack* and *unpack* methods indirectly using the *`uvm_field* macros, but their effect on performance can far outweigh their convenience [15].

You are not confined to using the *do_pack*/*do_unpack* approach. You can also implement a separate, custom converter class for a given type. With this approach, your SV transaction type does not need to inherit from *uvm_object,* provide a pack/unpack interface, nor conform to any other UVM requirement. For example:

SYSTEMVERILOG:
```
class packet; // no base class!
  rand cmd_t cmd;
  rand int unsigned addr;
  rand byte data[$];
endclass

import uvmc_pkg::*;
class pkt_cvrt extends uvmc_converter #(packet);
  static function void do_pack(packet t,
                               uvm_packer packer);
    `uvm_pack_int(t.cmd)
    `uvm_pack_int(t.addr)
    `uvm_pack_queue(t.data)
  endfunction
  static function void do_unpack(packet t,
                               uvm_packer packer);
    `uvm_unpack_int(t.cmd)
    `uvm_unpack_int(t.addr)
    `uvm_unpack_queue(t.data)
  endfunction
endclass
```

To use the custom converter, you specify it in one or more *connect* calls. The following *connect* call tells UVM Connect to use the *pkt_cvrt* converter for the *packet* transaction objects emitted by *p.out*.

```
uvmc_tlm#(packet,pkt_cvrt)::connect(p.out,"foo");
```

### B. SystemC Conversion Options

On the SystemC side, a separate converter class is typically defined. Any SC transaction object—preexisting or not— can be accommodated because there is no base class or interface requirement on the transaction class itself.

The following code implements a converter for the *packet* transaction class in SC. Note the use of overloaded << and >> operators for streaming the object's members to and from the inherited *packer* object.

SYSTEMC:
```
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;
  }
};
```

Using an optional convenience macro, the converter class definition for *packet* can be reduced to one line:

```
UVMC_UTILS_3 (packet, cmd, addr, data)
```

The '3' suffix in the macro name indicates that 3 members of the transaction are being packed and unpacked. The first macro argument is the transaction type, and the next three arguments are the transaction members. UVM Connect provides macros that can convert up to 20 members.

The macro expands into the converter definition shown above, i.e. what you would otherwise write on your own. The macro also defines an operator <<( ostream&) for your transaction so you can print the transaction to standard out.

```
cout << "Got transaction: " << trans << endl;
```

Although not recommended, another alternative is to implement *do_pack* and *do_unpack* methods in your SC transaction class. Documented examples with source code for this and the other conversion approaches can be found in the download kit.

## VI. UVM COMMAND API

The UVM command API gives SystemC users access to key UVM features in SystemVerilog. Currently, you can

- send UVM report messages, and set report verbosity.
- set and get UVM configuration, including objects.
- wait for UVM to reach a given simulation phase.
- raise and drop objections to phases to control UVM test flow.
- set type and instance overrides in the UVM factory
- print UVM component topology.

To enable use of the 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.

SYSTEMVERILOG:
```
module sv_main;
  import uvm_pkg::*;
  import uvmc_pkg::*;
  ...
  initial begin
    uvmc_init();
    run_test();
  end
endmodule
```

All calls 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.

### 1) *Send 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 within the UVM and SC hierarchies. As with natively issued UVM reports, all reports you send to UVM from SC 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.

```
SYSTEMC:
UVMC_INFO("SC_TOP/SET_CFG",
   "Setting cfg for SV producer",UVM_MEDIUM,"");
```

### 2) *Set and Get Configuration*

UVMC supports setting and getting integral, string, and object values from UVM's configuration database. This configuration can target SV components and SC models alike (provided the SC models know where to get their configuration).

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. You can later add as many configuration parameters to the config class without incurring any more overhead. With integrals 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 "max_len".

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

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

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

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

We can now configure an SV stimulus generator as follows:

```
SYSTEMC:
prod_cfg_sc 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);
```

The first argument is the name of the type in SV, which is passed to the UVM factory to create the object of that type in SV. The remaining arguments are the same as in SV: the context, field name, and the object we are setting. On the SV side, the object data is unpacked into a new factory-allocated instance and set into the UVM configuration database.

While not shown, the commands for setting integral and string configuration are nearly identical. In every case, the 'set' and 'get' syntax is much the same as in UVM.

### 3) *Phase Control*

The *uvmc_wait_for_phase* command lets a SC program 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.

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

uvmc_raise_objection("run","SC producer active");
  // produce data...
uvmc_drop_objection("run", "SC producer done");
```

### 4) *Factory Overrides*

You can also set UVM factory type and instance overrides from SystemC. Perhaps you are testing a SC-side model and want to drive it with a subtype of the SV stimulus generator that would normally be used. Once you make your overrides, you can then check that they "took" using some factory debug commands.

```
SYSTEMC:
uvmc_set_factory_type_override(
         "producer_t",      // old type
         "new_producer_t",  // new type
         "env.*");          // context
// UVM should report above overrides
uvmc_print_factory();
```

The above code tells the UVM factory to produce a *new_producer_t* object whenever an object of type *producer_t* is requested by any components hierarchically below a component named *env*. Both classes must be defined in SV, and the override type must be some extension of the overridden type.

### 5) *Printing UVM Topology*

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

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

cout << "UVM Topology:" << endl;
uvmc_print_topology();
```

The complete source code and helpful documentation for the above connect, converter, and UVM command examples can be found in the UVM Connect kit. Again, this is a free download and should run on all simulators.

## VII. MIXED-LANGUAGE CHALLENGES

In this final section we will describe some of the challenges we faced (and overcame) during development of UVM Connect.

### A. Local connections/binding proxies

To support vendor independence, the SC and SV engines needed to elaborate and start independently. Typically, SC is ready to go well before SV and reaches its elaboration phase before cross-language ports have a chance to be connected. Because all ports and sockets in both SC and SV UVM must be bound before end of elaboration, each *connect* and *connect_hier* call creates and binds a proxy to its cross-language counterpart. For example, when you call *connect* passing an initiator socket, UVM Connect will create a local target socket and bind the initiator to it, thus satisfying the connection policy in the local language. During UVM's elaboration phase, DPI-based communication between this proxy and its counterpart proxy in the other language is established.

**Figure 2: Local Proxies to Foreign Connections**

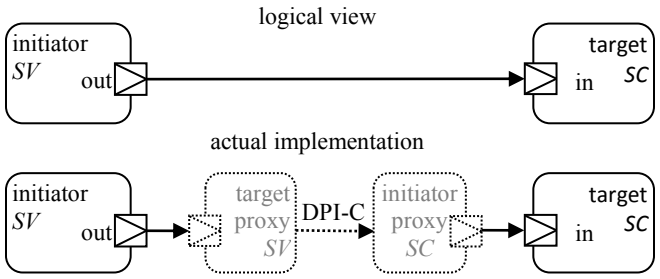


Figure 11. Transparent embedding of SV models in SC testbench

### B. Standard DPI-C only

The requirement for vendor independence meant we could not rely on vendor-specific solutions.

- We could not employ SV-on-top or SC-on-top solutions—although similar in each vendor, they are non-standard. The two language engines must come up independently (while still allowing vendor-proprietary solutions).
- We could not employ DPI-SC or other non-standard programming interfaces, nor call DPI export tasks from outside a DPI import task context.

  We had to provide TLM blocking semantics between SC and SV using only the standard DPI-C. However, DPI-C provides no explicit support for blocking calls from outside SV, except to allow a DPI-import task to call a DPI export task. We did not want to context switch when doing non-blocking communication between SC and SV.
- We had to be careful to use only data types supported by all three vendors when making DPI-C calls.

To solve this problem, each blocking call makes a non-blocking request to the other side, then waits on a native event. When the other side is ready with the response, the data is transferred and the event triggered via another non-blocking call. Thus, for TLM communication, the need for a formal C++ or SystemC-specific calling mechanism (i.e. DPI-C++ or DPI-SC) is mitigated.

### C. Multiple boundary crossings

A transaction can traverse the language boundary several times on its path to a final target. When a transaction makes its first crossing, we must create a new proxy transaction in the receiving language and unpack the contents of the original transaction into it. If the path of the transaction makes its way back to the originating language, we must not create a new transaction; we must use the same originating transaction object. If the transaction path makes yet another crossing, we must use the same proxy transaction object created during the first crossing.

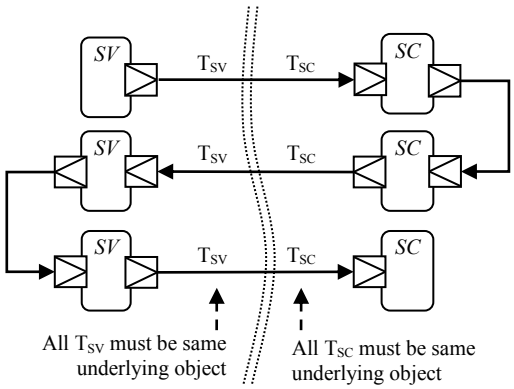


Figure 12. Transparent embedding of SV models in SC testbench

To solve this problem, we implemented a cache of outstanding transactions on each side of the language boundary, associating the original and proxy transactions with the same inter-language ID. The first language crossing establishes a new cache entry on each side, with both entries being associated with the same ID. When making subsequent language crossings, this ID is transferred along with the transaction data. The receiving language checks if there is a cache entry at that ID. If so, it retrieves the existing transaction object from the cache rather than allocate a new transaction. When the transaction execution is completed (according to TLM base protocol), it is removed from the cache.

### D. Multiple outstanding transactions and reentrancy

Multiple threads (SC or SV) may call into a blocking task, e.g. b_transport, over the same TLM connection. Multiple non-blocking transactions can be outstanding over the same connection as long as the request and responses do not overlap. This implies at most two outstanding transactions. However, temporal decoupling allows any

number of outstanding transactions as long as they aren't presented to the target in a way that violates the first rule.

Above all, UVM Connect must act as a transparent bridge, reflecting verbatim what it receives on one side to the other side. Thus, it cannot, for example, rely on the TLM payload event queue (PEQ), because that precludes the user application from providing that functionality, perhaps with enhancements.

To solve this problem, every TLM 2.0 blocking interface call causes a new thread to handle each request on the receiving side. For blocking and non-blocking transactions, the caching mechanism used for multiple boundary crossings is also used to track multiple concurrent transactions over the same socket.

### E. *Avoiding Stack Overflow*

When calling another language, the calling thread's stack is used in the foreign language, which may cause stack overflow. When that happens, often there is little to no clue as to why the simulator crashed or is behaving erratically. Control over stack size for SV threads is vendor dependent, but standard in SC using sc_spawn_options. We increased the default stack size used for spawned SC threads (to handle blocking calls mentioned previously) to avoid stack overflow in our test cases. In practice, heavy use of local variables and deep function call chains in your TLM method implementations may require you increase the stack size. For threads originating from UVM Connect, e.g. those spawned to handle blocking transport calls, you can specify a different default stack size via a global variable.

## VIII. SUMMARY

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. These features facilitate convergence of system-level activities that might otherwise serve in isolation. It also provides another compelling argument for applying the principles and purpose of object-oriented programming languages and the TLM interface standard. Such models can now also be integrated in both native and mixed-language environments without modification.

Since its release in early 2012, UVM Connect has been downloaded and used on real-world testbenches by companies using any or all three major simulator vendor's tools. Changes and improvements have been submitted by some of these companies and will be incorporated into future releases. The changes will make UVM Connect work better with others' simulators, improve ease-of-use, enhance performance, and increase compliance with the TLM standard. We encourage more of this collaborative effort to better align UVM Connect to real-world usage.

The latest library is available for download at verificationacademy.com [16] and uvmworld.org [17]. It is distributed under the Apache license, meaning it is free for you to use, modify, and share. If you would like to have your improvements included in an update release, feel free to contact the author to discuss how to get that done.

## IX. REFERENCES & RESOURCES

Below is a partial list of sources for information on SystemC, SystemVerilog, UVM, and mixed-language communication.


[1] UVM Connect download: See http://uvmworld.org/contributions area and https://verificationacademy.com/cookbook/UvmConnect for latest release. The latter reference contains video tutorials, online examples, and other documentation.

[2] Foster, H. (2010). *The 2010 Wilson Research Group Functional Verification Study*. See http://tinyurl.com/44nyp5r

[3] C Standard Programming Language; a good starting point is http://en.wikipedia.org/wiki/C_(programming_language)

[4] *C++* programming language*;* a good starting point is http://en.wikipedia.org/wiki/C%2B%2B

[5] *Standard SystemC Language Reference Manual*; IEEE 1666-2011; See http://standards.ieee.org/getieee/1666/download/1666-2011.pdf

[6] *IEEE Standard for SystemVerilog-Unified Hardware Design, Specification, and Verification Language*; IEEE 1800-2009; December 11, 2009; see http://www.systemverilog.org

[7] *Universal Verification Methodology;* reference manual and library download: http://www.accellera.org/downloads/standards/uvm/

[8] Accellera Verification Intellectual Property (VIP) Technical Subcommittee. http://www.accellera.org/activities/committees/vip

[9] *Accellera and Open SystemC Initiative Announce Plans to Unite*. Accellera. June 22, 2011. See http://tinyurl.com/97bw446

[10] Edelman, R., Glasser, M., Saha, A., Yin, H. "*Inter-language function calls between SystemC and SystemVerilog,*" DVCon 2007, San Jose, CA. See http://dvcon.org/2007/ses5.html

[11] Aynsley, J. *"SystemVerilog Meets C++: Re-use of Existing C/C++ Models Just Got Easier"* Design & Verification Conference 2010, San Jose, CA. See http://www.doulos.com/knowhow/sysverilog/DVCon10_dpi_paper

[12] Grover, V. *"VCS Built-in TLI connectivity for UVM to SystemC TLM 2.0"* Sep. 20, 2012. Search for at http://www.vmmcentral.org . TLI is not open-source and believed to run only on VCS at this time.

[13] Cadence Design Systems.. *UVM ML*. See contribution at http://www.uvmworld.org. The contribution, while open source, is not complete. The code needed for cross-language communication is not included and is believed to run only on Incisive.

[14] Kohli, A. et al; Mantis 3087: *Improve interaction between SystemC, C++ and DPI*; http://www.eda.org/svdb/view.php?id=3087

[15] Erickson, A."*Are OVM/UVM Macros Evil? A Cost-Benefit Analysis" Best Paper Award,* DVCon 2011, San Jose, CA. See http://verificationacademy.com/uvm-ovm/MacroCostBenefit

[16] Verification Academy. http://verificationacademy.com A free site for learning about almost any verification topic.

[17] UVM World. http://uvmworld.org

[18] UVM Cookbook - http://verificationacademy.com/uvm-ovm