UVM and SystemC Transactions - An Update

David Long

Doulos, Ringwood, UK.

John Aynsley

Doulos, Ringwood, UK.

***Abstract*-: This paper revisits the differences between Transaction Level Modeling (TLM) components implemented in SystemC and SystemVerilog UVM and the requirements of an efficient mechanism for passing transactions between them. It examines and compares the approaches taken by the most popular open-source solutions for connecting SystemC to UVM: UVM Connect [1] and UVM-ML [2]. It demonstrates how UVM Connect and UVM-ML provide a simple solution for insertion of an untimed or "loosely timed" SystemC model into a UVM environment. The paper goes on to explore additional features required by more accurate "approximately timed" SystemC models and looks at how these may be utilized in UVM Connect and UVM-ML. It also suggests possible approaches for unsupported features such as pipelined transactions and "payload event queues" in SystemC. The paper concludes by considering how the adoption of the proposed UVM-SystemC [3] standard might change our recommended approach, and looking further ahead, the implications of a multi-language standard supported by all tool vendors.**

I. Introduction

In many SoC designs, the embedded processors are blocks of IP that each have a SystemC TLM model. The testbenches for hardware blocks and the complete SoC are typically written in SystemVerilog. To verify the operation of hardware blocks, embedded software and the complete SoC, it is natural to want to connect together the various SystemC and SystemVerilog models. Unfortunately, even though SystemVerilog simulators have supported co-simulation of SystemC code for many years, passing transactions between models written in different languages is still not straight-forward.

In 2010, Doulos presented a paper [4] at DVCon looking at passing transactions between a SystemC model and a SystemVerilog testbench. It demonstrated that if you wanted your code to work consistently across multiple vendors' tools, your best bet was create your own string-based mechanism using DPI.

Today, the Universal Verification Methodolgy (UVM) has become the standard approach for writing SystemVerilog testbenches. UVM contains SystemVerilog implementations of SystemC's TLM1 and TLM-2.0 classes and major EDA vendors offer open-source solutions for passing transactions between the languages. An Accellera Systems Initiative (ASI) working group has recently proposed a SystemC implementation of UVM [3] and another ASI working group is currently looking at standard simulation interfaces for multi-language models. So is it any easier for end users to pick a mechanism for sending transactions between SystemC and SystemVerilog? Judging from the questions we regularly get asked: Apparently not!

II. Background

This section is aimed at readers who may not have an in-depth knowledge of UVM or SystemC. It provides a brief summary of the structure and operation of a UVM environment, a SystemC transaction level model and the issues of communication between them.

The standard UVM package provides a set of base classes that can be extended to create the building blocks of a complex verification environment. Communication between components takes place at a transaction level. Each transaction in a typical UVM environment (e.g. for a device-under-test accessed via a memory-mapped bus) might represent a single read or write bus-cycle or even a burst of data corresponding to many bus cycles. TLM communication has several advantages for UVM components compared to the wires used within traditional RTL testbenches:

1) transactions provide a higher level of abstraction that masks low-level details and hence assists reuse of components between testbenches;
2) the number of simulation events associated with each transaction (often just one event) is many times less than the corresponding number of RTL "pin wiggles" which leads to improved simulator performance;
3) transaction classes for basic tests may be extended to easily produce transaction classes for more specialized tests (and components that only accessed the fields in the original transaction classes would not require any modifications to continue working correctly with the extended transaction classes).

The TLM approach in UVM has been adapted from the TLM1 and TLM-2.0 approaches that are part of the SystemC standard [5]. As in SystemC, a transaction in UVM is initiated by calling a method provided by a "port" that is part of a component. The port is connected to a matching "export" on another component. If the parent component of the export implements the called method (rather than passing it on by calling a corresponding method in a child component) the export is known as an "imp" The method calls may either by "blocking" (the call may wait before returning so is implemented by a task) or non-blocking (the call returns immediately so may be implemented by a function). The port and export are parameterized classes that provide a mechanism to check that the connections between components are valid. The most common UVM TLM connection is between a "sequencer" component generating a stream of transactions and a "driver" that translates each transaction into a corresponding set of waveforms to drive the pins of the device under test. The other common TLM connection is the "analysis port/export" used to send transactions from a monitor (e.g. looking at bus activity) to a checker or scoreboard. An example showing the connections between components in a typical UVM environment, including the TLM ports and exports, is given in Figure 1.

The first TLM library ("TLM1") released by OSCI (Open SystemC Initiative – now part of ASI) did not meet the requirements for high-speed models and provided limited support for model interoperability. A second library ("TLM-2.0") was subsequently developed which took a different approach to address these limitations. A fundamental difference between TLM1 and TLM-2.0 is the mechanism used to transfer the data within a transaction: with TLM1 the transaction object is copied or passed by reference at each "hop" between initiator and target; with TLM-2.0 the transaction uses a standard "generic payload" class which contains a pointer to the actual data array (a block of bytes). The generic payload enables transactions to be sent efficiently across multiple hops – the data array pointer is used to access the actual data only once the transaction has reached its final destination. The SystemC standard also defines a base protocol for TLM-2.0 to encourage interoperability of models from different providers ("IP Vendors"). At each step of the base protocol, information may be sent either from the initiator to the target or back from the target to the initiator. TLM-2.0 connections therefore require two port/export pairs – one for the forward path and another for the backward path. For convenience, the initiator port/export are grouped together in an "initiator socket" while a "target socket" does the same for the target's port/export. UVM supports TLM-2.0 communication between components but there are some differences compared to SystemC, which are discussed below.

The UVM implementation of TLM1 and TLM-2.0 is based on SystemVerilog versions of the relevant SystemC classes. Fundamental differences between C++ and SystemVerilog mean that the UVM class hierarchy and certain operations have to use a different approach. The SystemC classes for TLM communication make use of multiple inheritance of C++ virtual interface classes. This mechanism was not provided by SystemVerilog when UVM was created so comparing the source code for ports and exports against the SystemC ones will reveal significant differences. However, in practice, a typical UVM user does not need to be aware of these differences for TLM1 connections. For TLM-2.0 connections, UVM provides a separate initiator and target socket for blocking and non-blocking connections. In SystemC, initiator and target sockets support both blocking and non-blocking transactions (an initiator is allowed to switch between blocking and non-blocking behavior between transactions). TLM-2.0 handles conversion between blocking and non-blocking transport calls automatically using the simple target socket,

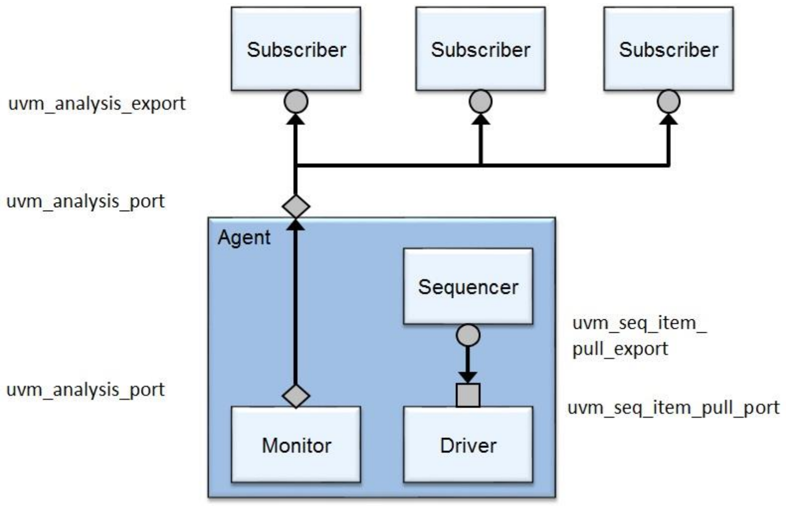

Figure 1 TLM connections within a typical UVM environment

which provides both blocking and non-blocking transport methods, only one of which actually needs to be implemented within the target. UVM does not provide this kind of automatic adaption.

In TLM-2.0, the generic_payload class defines get/set methods that must be used to access the members. In UVM, the generic payload data members are public and are declared as rand so the set/get methods are not required, although they are still provided. The UVM approach makes sense in the context of constrained random transaction generation; the UVM generic payload includes a set of default constraints to generate transactions with reasonable attribute values. (In contrast, the SystemC TLM-2.0 standard was developed with deterministic execution in mind, i.e. running system software.) Also, the UVM generic payload extends uvm_sequence_item, as would be the case for most other UVM transaction types, and defines methods for copying, comparing, and printing the payload contents.

Unlike C++, SystemVerilog does not support pointers. Objects are uniformly accessed by reference, and SystemVerilog provides automatic garbage collection. Because of these language differences, the UVM generic payload does not provide methods for explicit memory management (i.e. acquire, release, reset).

Unlike SystemC, where the data and byte enable array attributes are pointers, in UVM they are dynamic arrays. This makes the transaction data part of the transaction object itself, which is a difference in philosophy from TLM-2.0.

Like TLM-2.0, UVM supports adding generic extensions to the generic payload class. However, because SystemVerilog has no support for function templates, extensions have to be accessed by passing the extension ID as an argument to the set/get/clear_extension methods.

III. INTEGRATION OF SYSTEMC TLM MODELS IN UVM

Given the similarities between the TLM classes in UVM and SystemC, it is tempting to believe that integrating a SystemC TLM model in UVM would be a trivial exercise. All current simulators support mixed-language designs, and specifically permit SystemC modules to be instantiated directly from SystemVerilog. All simulators support the ability to make pin-level connections across languages, but unfortunately there is no standard for procedural (TLM) communication between SystemVerilog and SystemC aside from the DPI. The native SystemVerilog DPI does not explicitly support SystemC. However, it is possible to use the standard DPI with SystemC in the sense that SystemC is just another C++ application. SystemC modules typically provide a procedural interface either by implementing a SystemC interface or by offering a SystemC *export*. Using this procedural interface requires the ability to make C++ method calls, which is not possible using the standard DPI. In [4] we discussed ways to overcome this method calling limitation but there is also a limitation that applies to the transaction class itself. SystemVerilog DPI supports copying of “simple” types across the language boundary but this does not apply to either simple transaction classes or the generic payload: in order to copy them, they must first be converted to a format that is supported by DPI. UVM provides a mechanism that can be used for this conversion: the pack and unpack functions translate a UVM transaction class contents to/from a dynamic array of bit (or byte or int) – such arrays can be accessed in both SystemVerilog and C++ using the DPI. However, writing the code for this translation is typically a manual process that requires knowledge of how to use the DPI functions. The UVM Connect and UVM-ML are designed to automate this translation process.

IV. OVERVIEW OF UVM CONNECT

UVM Connect has been developed by Mentor Graphics but is released under an Apache 2 license. It uses SystemVerilog DPI features to communicate between SystemVerilog and SystemC and so can be used with any simulator that provides support for the standard DPI (as defined in IEEE 1800-2012). It is essentially a library of components that may be used to pass TLM1 and TLM-2.0 transactions between SystemC and SystemVerilog models. It also includes a “UVM Command API” that provides synchronization of SystemC processes with UVM phases and allows SystemC to raise and drop objections to prevent the current UVM phase from ending prematurely. It is intended to be relatively simple to use (although it is expected that users have some knowledge of the SystemC, SystemVerilog, UVM and TLM standards). It is well-suited to the application mentioned in the introduction (integrating an existing SystemC TLM model into a UVM environment) since the UVM Connect features require only minor changes to the existing code. A user guide, examples and compiler scripts for QuestaSim, Incisive and VCS are included in the download (these currently only support Linux versions of the tools).

The first requirement for integration of SystemC and SystemVerilog TLM communication is to connect the SystemC and SystemVerilog components. The UVM Connect library provides a set of connect and connect_hier functions for this purpose. These functions are listed in Figure 2– note that the SystemVerilog requires differently named parameterized functions for TLM1 and TLM-2.0 whereas SystemC can make use of overloaded function templates. The lookup argument is a string that uniquely identifies the port/export:

The second requirement for SystemC and SystemVerilog TLM communication is convert the transaction class to and from a stream format that can be used across the language boundary. The (optional) CVRT parameter in the

SystemVerilog TLM-2.0
```
uvmc_tlm #(T, CVRT)::connect (port, lookup);
uvmc_tlm #(T, CVRT)::connect_hier (port, lookup);
```

SystemVerilog TLM1
```
uvmc_tlm1 #(T, CVRT)::connect (port, lookup);
uvmc_tlm1 #(REQ,RSP,CVRT_REQ,CVRT_RSP)::connect (port,lookup);
uvmc_tlm1 #(T, CVRT)::connect_hier (port, lookup); TLM1 uni
uvmc_tlm1 #(REQ,RSP,CVRT_REQ,CVRT_RSP)::connect_hier (port,lookup);
```

SystemC TLM1 and TLM-2.0
```
uvmc_connect (port, lookup);
uvmc_connect <CVRT> (port, lookup);
uvmc_connect_hier (port, lookup);
uvmc_connect_hier <CVRT> (port, lookup);
```

Figure 2 UVM Connect Functions

connect functions in Figure 2 refer to a converter class that performs this operation. TLM-2.0 transactions using a generic payload (with no extensions) do not require a user-defined converter – UVM Connect provides built-in converter that is automatically applied. In a UVM environment, transactions passed over TLM1 or analysis ports/exports are often classes derived from the uvm_sequence_item base class. These classes will have inherited virtual do_pack and do_unpack functions – providing a definition for these classes is all that is required to enable them to work with UVM Connect. UVM provides a uvm_packer class that can be used within these functions to manage packing and unpacking for commonly-used transaction member types. SystemVerilog classes that are not derived from uvm_sequence_item require a custom converter class (derived from uvmc_converter) to be defined with static do_pack and do_unpack functions. SystemC transaction classes also require a custom converter class. However, in this case, the custom converter may be created as a template specialization of the uvmc_converter class for each required SystemC transaction type, with overridden do_pack and do_unpack member functions.

To support synchronization between SystemC processes and UVM phases, the UVM Connect Command API provides the methods shown in Figure 3. To enable the UVM Connect Command API layer, the top level SystemVerilog module should call uvmc_init from an initial block.

V. Overview of UVM-ML

The UVM-ML open architecture library has been developed by Cadence and like UVM-Connect, it is released under an Apache 2 license. It also uses SystemVerilog DPI features to communicate between SystemVerilog and SystemC and so can be used with any simulator that provides support for the standard DPI (as defined in IEEE 1800-2012). However, it is intended to fulfill a more complex set of requirements than UVM Connect: it includes frameworks for multiple languages together with associated adapters that manage the communication with its multi-language backplane. A framework is the implementation of a verification methodology using a particular language, for example UVM in SystemVerilog. An adapter provides the bridge between a framework and the multi-language backplane, which allows multiple frameworks to communicate independently. Much of the code, documentation and examples that make up UVM-ML is concerned with co-simulation of SystemVerilog and *e*. In this paper, we will only be considering the features of UVM-ML that are related to co-simulation of SystemVerilog UVM and SystemC.

Whereas UVM Connect provides a mechanism to connect standard SystemC models to a SystemVerilog UVM environment, UVM-ML provides a C++ implementation of various UVM classes that enable SystemC models to be written as UVM components: classes derived from the uvm_component base class with member functions that are automatically synchronized to the UVM phase scheduler. This requires a customized version of the ASI SystemC

```
uvmc_wait_for_phase(ph_name, UVM_PHASE_STARTED);
uvmc_raise_objection(ph_name, name(), "...");
uvmc_drop_objection(ph_name, name(), "...");
```

Figure 3 UVM Connect Synchronization Methods

library (the source code is provided as part of the UVM-ML download). A SystemC hierarchy of these uvm_component classes may be created within a SystemVerilog uvm_component hierarchy. It is also possible to use the customized SystemC libraries provided with each supported simulator. However, this requires the top of the SystemC hierarchy to be either an sc_module class or the sc_main function. To avoid C++ name conflicts, the UVM-ML code uses two namespaces: uvm and uvm_ml.

The connection of ports and exports across the language boundary requires port registration functions in SystemC and SystemVerilog. The registration functions provided by UVM-ML for standard TLM1 and TLM-2.0 connections are given in Figure

A connect function must also be called in either SystemC or SystemVerilog to make the connection between ports and exports or between initiator and target sockets: The connect function takes two string arguments that give the hierarchical path name of each port/export/socket.

```
bit res = uvm_ml::connect("sys.prod.isocket",c.target_socket.get_full_name()); //SV
uvm_ml::uvm_ml_connect("sys.u.my_outport" , sc_env.cons.my_export.name ()); //SC
```

UVM-ML provides automatic conversion between SystemC and SystemVerilog for TLM-2.0 transactions using the standard tlm_generic_payload class. TLM1 transaction classes are derived from the uvm_object base class (in both SystemC and SystemVerilog). Each TLM1 transaction class should provide an overridden do_pack and do_unpack function. The SystemC implementation of the uvm_packer class has overloaded streaming operators (<< and >>) for primitive and common types (e.g. vector<int>) as well as for standard SystemC data types.

When using the UVM-ML SystemC library, a call to the wait function in a SystemC process will automatically synchronize the SystemC time with the SystemVerilog time. If a different SystemC library is used, explicit synchronization between the SystemC and SystemVerilog frameworks can be achieved by calling the uvm::synchronize function from within a SystemVerilog process – this broadcasts the current time to the other frameworks

VI. INTEGRATION OF TLM1

This section compares the implementations of a mixed SC-SV TLM1 producer-consumer model in UVM Connect and UVM-ML. The producer is a SystemC model while the consumer is a SystemVerilog UVM component. Since our main focus here is on the mechanism for passing transactions between SystemC and SystemVerilog, we have simplified the model by omitting other components that would usually be found in a typical UVM environment (e.g. drivers, sequencers, monitors, etc). The model architecture is shown in Figure 6. To highlight the differences between UVM Connect and UVM-ML, we have built and run the code on the same simulator – in this case Mentor Graphics QuestaSim version 10.4b running on a specially created virtual machine with a standard version of CentOS Linux 7 64-bit installed. The version of UVM Connect used for the tests was uvmc-2.3.0 with UVM 1.1d. It was compiled using the Makefile supplied in the download. The version of UVM-ML was 1.5.1 with UVM 1.1d and SystemC 2.3.0 (we used the version supplied with UVM-ML since we could not get the “portable” adapter and the QuestaSim version of SystemC to work together correctly).

SystemVerilog TLM-2.0
```
uvm_ml::ml_tlm2 #(T)::register(socket);
```

SystemVerilog TLM1
```
uvm_ml::ml_tlm1#(T1,T2)::register(port,T1_name,T2_name);
uvm_ml::ml_tlm1#(T1,T2):register_directed(port,dir,T1_name,T2_name);
```

SystemC TLM-2.0
```
ml_tlm2_register_initiator(this,socket,i_socket_name,this_name);
ml_tlm2_register_target(this,socket,t_socket_name,this_name);
```

SystemC TLM1
```
uvm_ml::uvm_ml_register(&port);
```

Figure 4 UVM-ML register functions

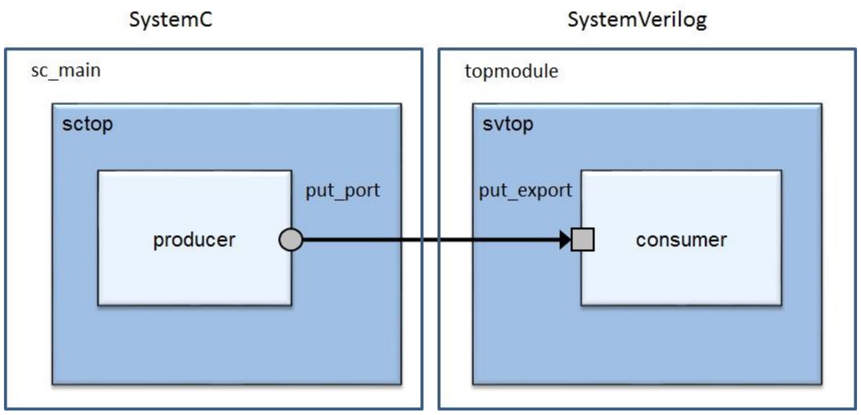

Figure 6 Demonstration TLM1 Model Architecture

The significant features of the SystemC producer used with UVM Connect are illustrated by the code in Figure 5. Note that the transaction class trans_t is not derived from any “special” class and that it includes a vector. The translation of transactions to and from a format that can cross the language boundary is performed by the do_pack and do_unpack functions respectively which are static members of the template specialization of the built-in uvmc_converter class. A template specialization of uvmc_converter must be defined for each type of transaction that will be used with UVM Connect but this is easily done since the uvmc_packer class has overloaded streaming operators that allow packing/unpacking of each type in trans_t (including the vector). The producer run function calls the TLM put task provided by its tlm_blocking_put_if port to send a series of transactions to the consumer.

```
#include "uvmc.h"
enum dir_t { READ, WRITE };

class trans_t {
public:
  int addr;
  std::vector<unsigned char> data;
  dir_t dir;
};

class producer : public sc_core::sc_module {
  public:
  sc_core::sc_port<
      tlm::tlm_blocking_put_if<trans_t> >
                                    put_port;
  producer (sc_core::sc_module_name nm) :
                     put_port("put_port") {
    SC_THREAD(run_proc);
  }
  SC_HAS_PROCESS(producer);
  void run() {
    trans_t * t;
    sc_core::wait(10, SC_NS);
    t = new trans_t;
    for (int i = 0; i < 8; i++) {
      ...
      put_port->put(*t);
    }
  }
  void run_proc() {
    uvmc_raise_objection("run");
    run();
    uvmc_drop_objection("run");
  }
};
```

```
  template <>
  struct uvmc_converter<trans_t> {
    static void do_pack(const trans_t &t,
                 uvmc::uvmc_packer &packer){
      packer << t.addr << t.data << t.dir;
    }
    static void do_unpack(trans_t &t,
                 uvmc::uvmc_packer &packer){
      packer >> t.addr >> t.data >> t.dir;
    }
  };

  int sc_main(int argc, char* argv[])
  {
    producer producer1("producer1");
    uvmc::uvmc_connect(producer1.put_port,
                       "put_port");

    sc_start();

    return 0;
}
```

Figure 5 UVM Connect SystemC producer

The producer run_proc thread raises an objection in the UVM run phase before calling a run function. When the run function returns, run_proc drops the objection which allows the UVM phase scheduler to move on to the next phase and end the simulation. The run and run_proc functions could have been combined into a single function but keeping them separate is an easier approach when reusing an existing SystemC model : run_proc is the only part of the producer class that depends on UVM Connect.

The SystemVerilog code for the UVM Connect consumer is given in Figure 7. Note that the transaction class is derived from uvm_sequence_item and has overridden do_pack and do_unpack (here we are using macros provided by UVM for packing and unpacking but we could have alternatively called the corresponding uvm_packer pack_*/unpack_* functions). The put_export member of the consumer class is a uvm_blocking_put_imp. This places a requirement on the consumer to provide an implementation of the put task. The trans_t and consumer class are standard UVM code. The UVM Connect features are used in the test that instantiates the consumer and the top level model. The connect_phase function in the test calls the UVM Connect uvmc_tlm1 #(T)::connect function to connect the consumer's put_export to the SystemC producer port. The parameter T is trans_t (the type of transaction) and the function string argument matches the name given to the producer's port when it was created. An initial block in the

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

class trans_t extends uvm_sequence_item;
  `uvm_object_utils(trans_t)
  typedef enum {READ,WRITE} dir_t;

  rand int addr;
  rand byte data[$];
  rand dir_t dir;

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

  virtual function void do_pack(
                   uvm_packer packer);
    `uvm_pack_int(addr)
    `uvm_pack_queue(data)
    `uvm_pack_enum(dir)
  endfunction: do_pack

  virtual function void do_unpack(
                    uvm_packer packer);
    `uvm_unpack_int(addr)
    `uvm_unpack_queue(data)
    `uvm_unpack_enum(dir, dir_t)
  endfunction: do_unpack
endclass: trans_t

class consumer extends uvm_component;
  uvm_blocking_put_imp #(trans_t,consumer)
                         put_export;

   `uvm_component_utils(consumer)

  function new(string name,
               uvm_component parent=null);
    super.new(name,parent);
    put_export = new("put_export",  this);
  endfunction

  task put (trans_t p);
    ...
  endtask

endclass
```

```
  import uvm_pkg::*;
  import uvmc_pkg::*;
  `include "uvm_consumer.sv"
  class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  function new(string name,
               uvm_component parent);
    super.new(name,parent);
  endfunction
  consumer cons;
  function void build_phase(uvm_phase phase);
    cons = consumer::type_id::create(
                            "cons",this);
  endfunction
  function void connect_phase(
                      uvm_phase phase);
    uvmc_tlm1 #(trans_t)::connect(
              cons.put_export, "put_port");
  endfunction
  endclass: my_test


  module topmodule;
  initial begin
    uvmc_init();
  end


  initial begin
    run_test("my_test");
  end


endmodule
```

Figure 7 UVM Connect SystemVerilog consumer

topmodule calls the UVM Connect uvmc_init function to enable the synchronization features in the SystemC model (a call to wait and raising/dropping objections).

The UVM-ML code for the SystemC transaction, producer and top level is given in Figure 8. The first thing to notice about this example is that it includes the uvm.h and uvm_ml.h header files which allows the SystemC code to look much more like regular UVM! The transaction class is now derived from uvm_object while the producer is derived from uvm_component. Also notice the UVM_OBJECT_UTILS and UVM_COMPONENT_UTILS macros – UVM-ML includes a “factory” for creating objects and components which can be overridden just as in the standard version of UVM (although we are not using any of those features here). The SystemC UVM-ML uvm_object class has pure virtual do_pack, do_unpack, do_print, do_copy and do_compare functions so overriding them in our transaction class is mandatory. The uvm_packer class in UVM-ML overloads the streaming operators in a similar way to UVM Connect. The UVM_OBJECT_REGISTER macro that is called for the transaction class registers the type

```
#include "uvm.h"

enum dir_t { READ, WRITE };

class trans_t : public uvm::uvm_object {
public:
  UVM_OBJECT_UTILS(trans_t)

  trans_t() {
    data.reserve(8);  //allocate space
  }
  int addr;
  std::vector<unsigned char> data;
  dir_t dir;

  virtual void do_pack(
           uvm::uvm_packer &packer) const {
    packer << addr << data << dir;
  }
  virtual void do_unpack(
            uvm::uvm_packer &packer) {
    int i;
    packer >> addr >> data >> i;
    dir = static_cast<dir_t>(i);
  }

  virtual void do_print(ostream& os) const {
    ...
  }

  virtual void do_copy(
             const uvm::uvm_object* rhs) {
    ...
  }

  virtual bool do_compare(
        const uvm::uvm_object* rhs) const {
    ...
  }
};

UVM_OBJECT_REGISTER(trans_t)
```

```
class producer : public uvm::uvm_component {
  public:
  sc_core::sc_port<
        tlm::tlm_blocking_put_if<trans_t> >
        put_port;
  producer(sc_core::sc_module_name nm) :
         uvm_component(nm),
         put_port("put_port") { }
  UVM_COMPONENT_UTILS(producer)
  void run() {
    trans_t * t;
    sc_core::wait(10, SC_NS);
    ...
    put_port->put(*t) ... }
  };

#include "uvm_ml.h"
class sctop : public sc_core::sc_module {
public:
  producer prod;
  sctop(sc_core::sc_module_name nm) :
             sc_module(nm), prod("prod") {
    uvm_ml::uvm_ml_register(&prod.put_port);
  }
  void before_end_of_elaboration() {
    uvm_ml::uvm_ml_connect(
                prod.put_port.name(),
               "svtop.consumer.put_export");
  }
};
int sc_main(int argc, char** argv) {
  return 0;
}


UVM_ML_MODULE_EXPORT(sctop)
UVM_COMPONENT_REGISTER(producer)
```

Figure 8 UVM-ML SystemC producer

name with the UVM-ML backplane so that the corresponding SystemVerilog class can be accessed when the transaction is passed to the SystemVerilog framework.

The top level module constructor instantiates the producer and calls uvm_ml_register (to give UVM-ML access to the producer's port). Before the end of elaboration, it calls uvm_ml_connect to connect the producer port to the consumer export. The sc_main function is required by QuestaSim in order to build the SystemC framework but it does not instantiate the top level module or call sc_start (since the simulation will be run under the control of the SystemVerilog framework).The top level module and the producer component are registered with UVM-ML by calling the UVM_ML_MODULE_EXPORT and UVM_COMPONENT_REGISTER macros respectively.

The UVM-ML SystemVerilog consumer and top level are shown in Figure 9. The transaction class and consumer are standard UVM and are exactly the same as for the UVM Connect example. The svtop component creates the consumer in the build phase and then registers the consumer's export with UVM-ML at the end of the build phase (in the phase_ended callback function). The top level module sets up an array containing the names of the top level SystemC and SystemVerilog components. This is then passed as the first argument to the uvm_ml_run_test task to start the simulation. Note that the UVM run_test task is not called! UVM-ML automatically raises and drops objections around the SystemC component's run thread so no explicit raising or dropping of objections are required in this simple example. If the run_phase of the SystemVerilog components also raised and dropped objections, regular calls to uvm_ml::synchronize would be required from System,Verilog to ensure the simulation time in both frameworks remained in step.

We compiled the UVM Connect and UVM-ML versions of the producer-consumer model and ran simulations for each one under the same conditions. On comparing the results, we found the behavior for both was the same.

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

class trans_t extends uvm_sequence_item;
  typedef enum {READ,WRITE} dir_t;
  rand int addr;
  rand byte data[$];
  rand dir_t dir;

 `uvm_object_utils(trans_t)
 ...
endclass

class consumer extends uvm_component;
  uvm_blocking_put_imp #(trans_t, consumer)
    put_export;
 `uvm_component_utils(consumer)
  ...
endclass
```

```
import uvm_pkg::*;
`include "uvm_macros.svh"
 import uvm_ml::*;)
class svtop extends uvm_env;
  consumer cons;
  `uvm_component_utils(svtop)
  function void build_phase(uvm_phase phase);
    cons = consumer::type_id::create(
                  "consumer", this);
  endfunction
  function void phase_ended(uvm_phase phase);
    if (phase.get_name() == "build") begin
      uvm_ml::ml_tlm1#(trans_t)::
            register(cons.put_export);
    end
  endfunction
endclass

module topmodule;
initial begin
  string tops[2];
  tops[0]   = "SC:sctop";
  tops[1]    = "SV:svtop";

  uvm_ml_run_test(tops, "");
end
endmodule
```

Figure 9 UVM-ML SystemVerilog consumer

VII. Integration of TLM-2.0

A mixed SC-SV loosely-timed or untimed TLM-2.0 initiator-target model will use the blocking transport interface to pass transactions. Note that while the standard SystemC initiator and target sockets support both blocking and non-blocking transport, UVM sockets are either blocking or non-blocking – you cannot call b_transport and nb_transport throught the same UVM socket. The TLM1 producer-consumer model from the previous section can be converted to a TLM-2.0 version by replacing the blocking put port and blocking put export with a tlm_initiator_socket and a uvm_tlm_b_target_socket respectively. The transaction class will be replaced by tlm_generic_payload /uvm_tlm_generic_payload. Conversion between SystemC and SystemVerilog generic payload transactions is managed automatically by both UVM Connect and UVM-ML – it is not necessary to write functions such as do_pack or do_unpack.. It is important to understand that the contents of a generic payload are managed differently in a mixed SC-SV environment compared to a pure SystemC one. As discussed in section II, the SystemC and SystemVerilog UVM implementations of the generic payload are slightly different. Further, it is not legal to pass a reference to memory allocated by SystemC to SystemVerilog, or vice-versa. Therefore, whenever a generic payload gets passed across the language boundary, its contents must be copied to a different transaction object. With blocking transport in both UVM Connect and UVM-ML, the payload is copied from the initiator to the target at the start of the transaction and is then copied back from the target to the initiator at the end of the transaction – neither the initiator that calls b_transport or the target that implements b_transport need to be aware that the payload content has been copied.

The other changes that need to be made to convert the producer-consumer to TLM-2.0 is to replace the TLM1 connect and register functions given in Figure 2 and Figure 4 with the corresponding TLM-2.0 ones (we are not showing the code here due to lack of space).

Converting the producer-consumer example to use non-blocking transport could become far more complicated! Firstly, the TLM standard requires non-blocking transport to use a memory manager to allocate the memory for each payload object (usually from a managed "memory pool"). Secondly, the base protocol means a transaction could potentially consist of up to four phases. Each phase is associated with a corresponding call to nb_transport_fw or nb_transport_bw. The payload object may therefore have to be copied between the initiator and target, along the forward and backwards path, multiple times during each transaction. Each payload object must be returned to the memory pool when the transaction is complete, but not before. Great care is required when implementing the initiator and target to make sure the content and lifetime of the payload is managed correctly!

VIII. Limitations

UVM Connect and UVM-ML do not support the entire SystemC TLM1 and TLM-2.0 interfaces. Features such as the Direct Memory Interface would not be legal between SystemC and SystemVerilog. Other limitations appear to be a result of each vendor prioritizing their development efforts and concentrating on the features which they believe are most likely to be of interest to users: For example, UVM-ML does not support the use of generic payload transactions for TLM1 interfaces or hierarchical SystemC exports bound to a SystemVerilog port; while UVM Connect does not support instance-specific extensions of the generic payload.

A feature that is commonly used in approximately-timed SystemC simulations is the "payload event queue". This is a mechanism that simplifies the code for managing dynamic processes associated with multiple, overlapping transactions (such as may be found with an AXI bus model). The payload event queue is part of the SystemC standard (even though the language reference manual classifies it as one of the "TLM-2.0 utilities)" but it is not directly supported by either UVM Connect or UVM-ML. However, we found that it is still possible to include a payload event queue on the SystemC side, although it was necessary create a clone of the returned transaction data (using deep_copy_from) within nb_transport_bw before putting the transaction in the payload event queue. On the SystemVerilog side, we had to write code to create copies of pending transactions and to schedule each associated processes using uvm_event::p_trigger.

IX. Closer Integration of SystemVerilog and SystemC

The proposed UVM-SystemC library provides an interesting addition to the field of mixed SystemC-SystemVerilog simulation. UVM-SystemC provides a SystemC implementation of the core classes in the UVM library. This should make it easier to package a SystemC model as UVM verification IP (the term "Universal Verification Component" or UVC is often applied to such models). As with UVM-ML, UVM-SystemC includes a phase scheduler to ensure that component phase methods are called in the correct sequence, taking into account their position in the component hierarchy. The build_phase and connect_phases are called before the SystemC scheduler initializes and runs the simulation (while its status is still SC_ELABORATION) to enable deterministic configuration of components.

While UVM-SystemC describes a pin-level interface (sc_signal<T>) for RTL-level connections it does not currently address the issues of synchronizing phases and passing transactions between SystemC and SystemVerilog

UVM models (although it does provide an implementation of the uvm_packer class and streaming operators for do_pack and do_unpack). In fact, the initial release does not include any support for TLM-2.0 (primarily to avoid the inconsistencies between the UVM version and the SystemC standard). Synchronization and passing TLM-2.0 transactions between UVM-SystemC and SystemVerilog UVM will therefore still require the use of an additional library such as UVM Connect or UVM-ML. Since UVM-ML and UVM-SystemC define many of the same classes and both also use a "uvm" namespace, care would be required in building an application that used both libraries to avoid compiler name clashes or linker errors.

A standard solution to synchronization and transaction passing between SystemC and SystemVerilog TLM models might come from the ASI Multi-Language working group. This group has been looking at the issues of multi-language UVM and TLM models since 2012. It has generated a list of requirements and spawned a sub-group to work on a "skeleton" code implementation. However, at the time of writing (early 2016) there has been little other visible activity recently within the main group.

X. Conclusions

UVM Connect and UVM-ML both provide portable solutions for passing transactions between SystemC and SystemVerilog using SystemVerilog simulators from the three major EDA vendors. It is no longer necessary to create your own solution based on DPI and strings! We used Mentor Graphics QuestaSim for the examples presented here but it should be possible to re-run them on other vendors' tools by making only slight changes to the top level SystemC and SystemVerilog modules. Both UVM Connect and UVM-ML come with multiple examples showing how to do this.

We found that when connecting an existing SystemC model to a UVM environment, it was easier to get everything built and working correctly with UVM Connect since it only required us learning how to use a few basic constructs. On the other hand, UVM-ML was more complicated to learn but allowed us to create SystemC models in the same style as regular UVM, so in that sense it is more like UVM-SystemC. If you also need to include *e* code in your UVM environment, UVM-ML would be the obvious choice since it was specifically designed to support SystemVerilog, SystemC and *e* flavors of UVM!

Documentation, examples and scripts are provided with both UVM Connect and UVM-ML. There are also on-line tutorials for UVM Connect. However, we found that building the libraries and running the examples was still quite challenging (compared to installing the standard SystemC or UVM libraries). Building the UVM-ML library with QuestaSim was considerably more complicated than building UVM Connect (even taking into account that you would naturally expect Mentor's UVM Connect to be easier to build with the Mentor simulator). Extra attention was needed during the UVM-ML build to make sure the environment was set correctly and that the correct versions of SystemC and UVM were included. We would recommend that Cadence Incisive users wishing to investigate UVM-ML should use the version of the library that is part of their Incisive installation rather than attempting to build it for themselves.

References

[1] http://forums.accellera.org/files/file/92-uvm-connect-a-systemc-tlm-interface-for-uvmovm-v22/
[2] http://forums.accellera.org/files/file/65-uvm-ml-open-architecture/
[3] "UVM-SystemC language reference manual – draft", December 2015.
[4] J. Aynsley, "SystemVerilog meets C++: Re-use of existing C/C++ models just got easier", *DVCon*, 2010,
[5] "IEEE Standard for Standard SystemC Language Reference Manual", IEEE Std 1666-2011, January 2012