TLM-2.0 in SystemVerilog

Mark Glasser[1], Janick Bergeron[2]

[1]*Mentor Graphics Corporation*
Fremont, CA
`mark_glasser@mentor.com`

[2]*Synopsys, Inc.*
Mountain View, CA
`janick.bergeron@synopsys.com`

***Abstract*—Transaction-level modeling (TLM) is a methodology for building models at high levels of abstraction, those above RTL. TLM-2.0 is a library that contains classes that implements a methodology for building transaction-level models in systemC and connecting them together. It was developed by OSCI and released in 2009 and is now on its way to becoming an IEEE standard as part of IEEE-1666-2011. In order to support new use models in SystemVerilog, a translation of TLM-2.0 is now included in UVM. This paper will describe the translation of TLM-2.0 from SystemC to SystemVerilog.**

I. INTRODUCTION

TLM-2.0 was released as a standard by OSCI in July 2009. Since that time, it has become an important tool for building interoperable transaction-level models. The OSCI standard has been implemented in SystemC and to date has primarily been used to build architectural models of bus-based platforms in SystemC.

Recently, the SystemVerilog community has become interested in TLM-2.0. TLM-2.0 provides a number of features not available in TLM-1.0, including timing annotations, bidirectional blocking and nonblocking interfaces, and the generic payload. Although TLM-2.0 was primarily targeted for building transaction-level models of bus-based systems, those in the verification community are seeing that it can be used effectively in building testbenches as well. Additionally, connecting TLM-2.0-based SystemC models into a SystemVerilog environment is increasingly important in multi-abstraction verification environments.

Having originally been designed and implemented in SystemC, it was necessary to translate TLM-2.0 into SystemVerilog. Because of the differences in the two languages, it was not possible to translate the systemC implementation directly into SystemVerilog.

In this paper we will discuss the issues encountered when translating TLM-2.0 from SystemC to SystemVerilog and how they were addressed. We provide a tour of the major features on the SystemVerilog dialect of TLM-2.0. We will compare the SystemC and SystemVerilog versions of TLM-2.0 and show how to build essential structures using TLM-2.0. Finally, we will present an agenda for future verification-related work using the SystemVerilog version of TLM-2.0.

II. CHOOSING TLM-2.0 SUBSET

Translating any code between programming languages can be tricky. It's not simply a matter of converting syntax. The problem lies in matching the semantics of the target language to that of the source language. Douglas Hofstadter, in his book "Le Ton beau de Marot" [3] where he discusses natural language translation at length, says:

> ...I brought up some notions I had been exploring about "generalized translation". The basic idea was to conceive of translation as the faithful transport of some abstract pattern from one medium to another.

We may have an easier task translating computer code than natural language text, as Hofstadter discusses. Nevertheless, it is not the same as taking each line or function in SystemC and re-writing it in SystemVerilog. In translating TLM-2.0 from SystemC to SystemVerilog we tried to capture the essential abstract pattern of the code's meaning, i.e. the author's intent, and re-implement it in SystemVerilog. We were fortunate that the authors intent is clearly stated in the TLM-2.0 LRM [5] and in a reference implementation.

The essential features that we chose to focus on are:

- transport interfaces
- sockets
- generic payload

We felt that this was sufficient to implement the base protocol and enable the kinds of TLM-2.0 structures that are useful in implementing Verification IP and verification environments.

III. TRANSLATION ISSUES

SystemC and TLM-2.0 are implemented in C++. Translating some kinds of C++ code to SystemVerilog is straightforward, some kinds are not.

Like C++, SystemVerilog is an object-oriented language and supports classes and inheritance. However, SystemVerilog only supports single inheritance, whereas C++ supports multiple-inheritance. Multiple-inheritance had to be emulated in SystemVerilog to translate TLM-2.0 code that was implemented using multiple-inheritance in C++.

C++ provides a richer templating facility than SystemVerilog does. In particular, C++ supports function templates which are used in the implementation of TLM-2.0. SystemVerilog has no such feature.

The convenience sockets in TLM-2.0 rely on function pointers to register callbacks as implementations of the various interface functions. SystemVerilog does not support function pointers. It's possible to implement callback registration in convenience sockets as UVM supports the notion of callbacks. The best strategy for building convenience sockets which have function callbacks is still an area for investigation.

IV. IMPLEMENTATION DETAILS

In this section we will discuss the details of how TLM-2.0 is implemented in UVM.

Interfaces. TLM-2.0 contains two transport interfaces – a blocking and a nonblocking transport. The nonblocking transport interface provides bi-directional communication via a forward transport path, and a backward transport path. The nonblocking forward and backward transport paths perform essentially the same operation, just in reverse directions.

The transport functions are specified in SystemC as follows:

```
void b_transport(TRANS& trans,
                 sc_core::sc_time& t);

tlm_sync_enum nb_transport_fw(TRANS& trans,
                              PHASE& phase,
                              sc_core::sc_time& t);

tlm_sync_enum nb_transport_bw(TRANS& trans,
                              PHASE& phase,
                              sc_core::sc_time& t);
```

`TRANS` and `PHASE` are the transaction type and phase enum type, respectively, and these are specified as template parameters. The SystemVerilog translation of these interfaces is shown here:

```
task b_transport(T t, uvm_tlm_time delay);

function uvm_tlm_sync_e
    nb_transport_fw(T t, ref P p,
                    input uvm_tlm_time delay);

function uvm_tlm_sync_e
    nb_transport_bw(T t, ref P p,
                    input uvm_tlm_time delay);
```

In SystemC, the transaction, phase, and time are all passed by reference. That means the target can modify the values of those arguments and the new values will be reflected back in the calling function. This pass-by-reference is one of the key element of TLM-2.0. Whereas C++ uses an ampersand on the right hand side of a variable to denote a reference, SystemVerilog uses the keyword `ref` to do essentially the same thing. The phase enum is a scalar so we use the `ref` keyword to pass this arguments by reference. The transaction and time arguments on the other hand are objects. Whereas object variables are treated like any other scalar variable in C++, object variables in SystemVerilog are always references (i.e. pointers) to the object instance. When assigning or passing object variables in SystemVerilog, object handles are passed by value and the entire object is not copied. Passing an object handle by value is similar to pass by reference. That is, the recipient of the handle can modify the contents of the object and those changes will be reflected back to the calling function (or task). Thus it is not necessary to use the `ref` keyword for the transaction and time arguments. Had we supplied the `ref` keyword for the transaction object then that would mean the value of the handle itself could be changed. Another object could have been substituted for the one we passed in. That semantic is not equivalent to pass-by-reference in C++.

Ports, Exports, and Imps. Ports and exports are connectors through which control and data is transferred in a transaction-level communication. A call to an interface function is initiated on a *port*. A corresponding *export*, which is connected to the port, responds. Since the port is making the call it `requires` the interface function to be available. The export `provides` is the entity that makes the function available. A transaction-level connection can be threaded through the hierarchy in a chain of ports and exports. On one end of the chain is a port, which initiates a transaction. At the other end of the chain is an *imp* which supplies the implementation of the function. For more details on how ports, exports, and imps work see [2].

SystemC comes natively with ports and exports. That is, port and export templates are part of the SystemC library. Both the TLM-1.0 and TLM-2.0 SystemC libraries supply interface templates, classes derived from `sc_interface`, which are used to create ports and exports. SystemVerilog does not natively have the notion of a port. However, base port classes are provided by UVM. They were built to support TLM-1.0 and now are used also to support TLM-2.0.

Defining a port in SystemC is a matter of supplying the interface type to the port template. E.g.

```
sc_port <tlm_blocking_get_if < my_trans > > get_port;
```

The equivalent in SystemVerilog looks like this:

```
uvm_blocking_get_port #(my_trans) get_port;
```

In the TLM-1.0 library in SystemVerilog, a collection of port objects exists, each specialized with the interface type. Since SystemVerilog does not support multiple inheritance, specialized classes are necessary. Each specialized port/export class both extends and implements the interface. The SystemVerilog implementation of TLM-2.0 also contains a set of specialized port and export classes for the interfaces described above. Unlike in TLM-1.0, users will not directly instantiate and connect TLM-2.0 ports and exports. Instead, ports and exports are used to construct sockets. Sockets are the objects that are directly instantiated and connected to form TLM-2.0 connections between components.

Sockets. A socket is a combined port and export. A socket represents a bidirectional connection between the initiator and the target. For the blocking interfaces, the forward path is activated by calling the task `b_transport()`[1], and the return path is the annotated transaction and delay arguments of the task. For nonblocking interfaces, the forward and backward paths are implemented by the `nb_transport_fw()` and `nb_transport_bw()` functions respectively.

Depending on the type of socket, a socket either is a port, is an export or imp, is a port and has an export or

---

[1] blocking interfaces must be tasks since functions cannot block

imp, or is an export and has a port. For the nonblocking sockets, the initiator socket contains in implementation of `nb_transport_bw` and the target socket contains an implementation of `nb_transport_fw`. Table I shows the possible combinations.

	Blocking	**Nonblocking**
initiator	IS-A fw port	IS-A fw port; HAS-A bw imp
target	IS-A fw imp	IS-A fw imp; HAS-A bw port
pass-through initiator	IS-A fw port	IS-A fw port; HAS-A bw export
pass-through target	IS-A fw export	IS-A fw export; HAS-A bw port

TABLE I
SOCKET ORGANIZATION

IS-A refers to the object-oriented relationship of inheritance. if D IS-A B then D inherits from B. *HAS-A* refers to an association between objects. Details on relationships between objects can be found in [4]. Discussion of how to represent object relationships graphically using UML can be found in [1].

The connection mechanism for sockets is the same as for ports and exports since sockets are built upon ports and export. A proper connection is ensured through a combination of compile time and run time type checking. The base class from which ports and exports are derived, `uvm_port_base#(IF)` is the same base class from which sockets are derived. `uvm_port_base#(IF)` contains a the `connect()` method which is used to form a connection between two objects derived from `uvm_port_base` — i.e. ports, exports, imps, and sockets.

The function `uvm_port_base::connect()` is virtual and thus can be overridden by the derived class. For the nonblocking sockets that are combined ports and exports, the `connect()` function first calls `super.connect()` which properly connects the base object. `connect()` perform as a run-time check to ensure that the connection is valid. Finally, the subordinate object is connected. As an example of this process here is the implementation of `uvm_tlm_nb_initiator_socket::connect()`:

```
super.connect(provider);

if($cast(initiator_pt_socket, provider)) begin
  initiator_pt_socket.bw_export.connect(bw_imp);
  return;
end
if($cast(target_pt_socket, provider)) begin
  target_pt_socket.bw_port.connect(bw_imp);
  return;
end
if($cast(target_socket, provider)) begin
  target_socket.bw_port.connect(bw_imp);
  return;
end
c = get_comp();
`uvm_error_context(get_type_name(),
    "type mismatch in connect", c)
```

The call to `super.connect()` connects the base object, which in this case is a port. An initiator socket can only be connected to an initiator pass-through socket, a target pass-through socket, or a target socket. The series of `$cast()` calls checks to see which one of those it is and makes the appropriate connection of the subordinate export. If the object we are attempting to connect to is none of those socket types then and error is emitted and no connection is made. The connection process is similar for other socket types.

Generic Payload. The generic payload is designed to represent a bus transaction. It contains address, data, and other elements needed to describe a concrete transaction on a bus. The main data structure is very simple: a collection of scalar and array variables.

Translating those to SystemVerilog is straightforward. The types used in the SystemC generic payload and their SystemVerilog equivalents are shown in table II. Unsigned ints in C++ are unsigned ints in SystemVerilog; Enumerated types in SystemVerilog are very similar to those in SystemC. The arrays defined as `char*` are translated to dynamic arrays of unsigned bytes. A `bool` in SystemC is a `bit` in SystemVerilog.

SystemC	**SystemVerilog**
`sc_dt::uint64`	`bit [63:0]`
`tlm_command`	`uvm_tlm_command_e`
`unsigned char*`	`byte unsigned[]`
`unsigned int`	`int unsigned`
`tlm_response_status`	`uvm_tlm_response_status_e`
`bool`	`bit`

TABLE II
TYPE TRANSLATION FOR THE GENERIC PAYLOAD

There are several key differences which are worth noting. First, the SystemVerilog `tlm_generic_payload` class is derived from `uvm_sequence_item`, whereas the SystemC class does not inherit from any other class. In SystemVerilog, this enables generic payload objects to be used as sequence items. Another difference is the use of the `rand` keyword. SystemC does not have an automated way of randomizing classes, but SystemVerilog does. Either on its own or when used as a sequence item, the generic payload can be randomized.

Extensions. The SystemC LRM says that any instance of the generic payload can have an arbitrary number of extensions, but only one of any specific type. So that means that we need to be able to identify each extension object by type and ensure that no more than one of any type is present for a generic payload object.

In SystemC, extension objects of type `T` are derived from a base object, `tlm_extension<T>`. The base object contains some virtual functions for managing the extensions object, `clone()`, `copy_from()`, and `free()`. In SystemVerilog, a similar `uvm_tlm_extension<T>` class is provided. It is derived from `uvm_object` which contains the interface for basic object management such as `copy()`, `clone()`, etc. A function for freeing an extension object is not necessary in SystemVerilog as garbage collection is handled automatically.

In systemC, a function template is used by the `get_extension()` method to obtain, from the `tlm_extension` base class, an identifier that uniquely identifies the extension type to be retrieved from the generic payload object.

```
class tlm_generic_payload {
  ...
  template <typename T>
  void get_extension(T*& ext) const
  {
    ext = get_extension<T>();
  }
  template <typename T> T* get_extension() const
  {
    return static_cast<T*>(get_extension(T::ID));
  }
  // Non-templatized version with manual index:
  tlm_extension_base*
    get_extension(unsigned int index) const
  {
    return m_extensions[index];
  }
  ...
}
```

Due to the lack of template functions in SystemVerilog, only the non-templatized version of the `get_extension()` is provided and the static function `ID` must be explicitly called. Extensions can then be easily attached to, and later retrieved from, a generic payload object instance.

```
class my_extension
  extends uvm_tlm_extension<my_extension>;
   ...
endclass

my_extension ext1 = new("ext1");
gp.set_extension(ext1);
...
$cast(ext1, gp.get_extension(my_extension::ID));
```

Extensions have to be considered when copying, comparing, and printing generic payload objects. The implementation of `do_copy()`, `do_compare()`, `do_print()` iterates over all attached extension instances in the generic payload object and calls the corresponding copy, compare or print method on the extension.

Managing Time. TLM-2.0 supports different timing accuracy models for transaction-level models written in SystemC. A key element to being able to implement these various timing models is the ability of decoupling the execution time of an initiator from its target. This allows a target to model delays and activity up to a point some time in the future without having to involve multiple simulation threads or the scheduler.

This temporal decoupling is accomplished by advancing the nominal simulation time for a transaction by incrementing the `sc_core::sc_time& t` argument on the blocking and nonblocking interface functions. The nominal transaction time is subsequently decreased when appropriate when simulation time physically advances.

SystemC models time values using the `sc_time` type. Literal time values are specified by constructing an instance of the `sc_time` class which requires a physical unit to be specified. Because time values are always specified with a physical unit, a target and an initiator will always interpret a time value of 1.5ns the same way.

```
void b_transport(tlm_generic_payload& trans,
                 sc_core::sc_time& t)
{
   ...
   sc_time delta(1.5, SC_NS);
   t += delta;
}
```

Unfortunately, SystemVerilog inherited its time specification system from Verilog. It uses a simple 64-bit integer value and variable. The current `` `timescale `` implicitly specifies the physical unit of those integer time values. SystemVerilog did add the ability to explicitly specify a time unit with a time literal. However, it is a simple compile-time syntactical artifice and the resulting simulation-time value is still converted to a pure integer value based on the current timescale.

This approach is fine as long as the time values remain within the same compilation unit. SystemVerilog does not allow the timescale to be changed within a compilation unit. Therefore, the (unit-less) time values will be consistently interpreted. However, once it becomes necessary to pass time values across components that are implemented in different compilation units (e.g. different modules or different packages), there is a definite possibility that they will interpret the integer time values differently, with catastrophic results. Using the SystemVerilog type was therefore not suitable to exchange transaction timing information between an initiator and a target.

The solution was to implement in SystemVerilog a time type similar to SystemC's `sc_time` type: `uvm_tlm_time`. Like SystemC's `sc_time`, it represents time values using a specific time resolution (femtoseconds by default) into an absolute canonical time value. It can convert this canonical time value to and from a time value in the current timescale. It can similarly be incremented or decremented by a time value in the current timescale.

```
task b_transport(uvm_tlm_generic_payload trans,
                 uvm_tlm_time t);
   ...
   t.incr(1.5ns, 1ns);
endtask

b_transport(gp, t);
#(t.get_realtime(1ns));
t.reset();
```

The `"1ns"` time literal value is used to identify the timescale of the caller's scope to the `uvm_tlm_time` class, which is itself potentially compiled using a different timescale. The implicit type conversion in SystemVerilog converts the literal into a scaled, unit-less real time value. The `uvm_tlm_time` class is then able to compute the timescale of the caller by multiplying this reference time value by `10E9`. Once the scale of the caller is known, it is a simple matter of dividing incoming time values or multiplying outgoing time values to/from the resolution of the internal canonical time value.

V. CROSS-LANGUAGE COMMUNICATION

An important use model for TLM-2.0 in SystemVerilog is communicating between SystemVerilog and SystemC models. Increasingly common is verification engineers wanting to use SystemC architectural models as reference models in RTL testbenches. Complex stimulus generators or other models that are useful in testbenches may also exist in SystemC. Rewriting models is precarious at best; it's not easy to exactly duplicate semantics between languages (a problem we have to address in translating TLM-2.0 into SystemVerilog!) and, even when semantic conversion is straightforward, ensure identical behavior is difficult. Rather than rewriting those models in SystemVerilog, it is usually preferable to connect the SystemC models in a SystemVerilog verification environment.

For performance reasons, all the TLM-2.0 interfaces pass transaction objects by reference. However, this presents a problem when passing transactions between language domains: it is not possible to pass handles between language domains and allow the handle to be dereferenced in a language domain other than the one in which it was created.

When passing arguments by value, we can hide the data conversion operation as part of the copy. No such hiding is possible when doing pass-by-reference. That is why objects cannot be passed between SystemVerilog and C using the SystemVerilog DPI. Fortunately, one can attempt to emulate pass-by-reference when an object is shared across a language boundary.

One way would be to have a coherency mechanism that would cause the sister object to be updated whenever an object is modified. This is potentially complex and inefficient. Implementing this mechanism would require a system of event notifications and conversions between the two languages.

Fortunately, TLM-2.0 has well-defined timing points that are the only time that objects shared between an initiator and a target can be effectively modified. These timing points identify when an object and its sister object need to be brought into coherency cross the language boundary. Furthermore, these timing points occur only when an interface function is called. At other times it is acceptable that that there may be differences. When an interface function is called, a corresponding call is made in the associated component across the language boundary. At that point the passed transaction object can be converted into the other language. This is essentially the same as a pass-by-value model.

VI. CONCLUSIONS

The TLM-2.0 standard as ratified by OSCI is not just a SystemC tool. It defines a methodology for building transaction-level components that is independent of language. We have shown how those semantics can be implemented in SystemVerilog. Further, a facility that implements a large part of the TLM-2.0 standard in SystemVerilog has been released in UVM.

VII. FUTURE WORK

The most obvious area for future work is to translate more of the elements that are in the OSCI standard into SystemVerilog. This primarily means the convenience sockets.

It is unlikely that the SystemVerilog dialect of TLM-2.0 will be used extensively to build transaction-level models for the purpose of architectural analysis or for stepwise refinement into RTL. This kind of activity is the domain of SystemC and will likely remain there for a long time. However, there is work to be done to invent and discover meaningful use models for verification that rely on TLM-2.0 semantics. For example, can the nonblocking interfaces be used to improve the sequencer/driver interface? Is there an advantage to using TLM-2.0 interfaces to connect monitors to scoreboards? These are only a small sample of the questions to be investigated concerning how to apply TLM-2.0 semantics to testbench architectures.

REFERENCES

[1] M. Fowler. *UML Distilled*. Addison-Wesley, third edition edition, 2004.
[2] M. Glasser. *The OVM Cookbook*. Springer, 2009.
[3] D. R. Hofstadter. *Le Ton beau de Marot: In Praise of the Music of Language*. Basic Books, 1997.
[4] J. Martin and J. Odell. *Object-Oriented Methods: A Foundation*. Prentice Hall, 1995.
[5] Open SystemC Initiative. *OSCI TLM-2.0 Language Reference Manual*, 2009.