Writing TLM2.0-compliant timed SystemC simulation models for SoCLib

Authors : Alain Greiner, François Pêcheux, Emmanuel Viaud, Nicolas Pouillon, Aline Vieira de Mello

A) Introduction

This document is still under development. It describes the modeling rules for writing TLM-T SystemC simulation models for SoCLib that are compliant with the new TLM2.0 OSCI standard. These rules enforce the PDES (Parallel Discrete Event Simulation) principles. Each PDES process involved in the simulation has its own local time, and PDES processes synchronize through messages piggybacked with time information. Models complying to these rules can be used with the "standard" OSCI simulation engine (SystemC 2.x) and the TLM2.0 protocol, but can also be used also with others simulation engines, especially distributed, parallelized simulation engines.

The interested user should also look at the general SoCLib rules.

B) Single VCI initiator and single VCI target

Figure 1 presents a minimal TLM-T system containing one single initiator, **my_initiator** , and one single target, **my_target** . The **my_initiator** module behavior is modeled by the SC_THREAD **execLoop()**, that contains an infinite loop. The call-back function **my_nb_transport_bw()** is executed when a VCI response packet is received by the initiator module.

Unlike the initiator, the target module has a purely reactive behaviour and is therefore modeled as a simple call-back function. In other words, there is no need to use a SC_THREAD for these simple target components: the target behaviour is entirely described by the call-back function **my_nb_transport_fw()**, that is executed when a VCI command packet is received by the target module.

The VCI communication channel is a point-to-point bi-directionnal channel, encapsulating two separated uni-directionnal channels: one to transmit the VCI command packet, one to transmit the VCI response packet.

C) VCI initiator Modeling

In the proposed example, the initiator module is modeled by the **my_initiator** class. This class inherits from the standard SystemC **sc_core::sc_module** class, that acts as the root class for all TLM-T modules.

The initiator local time is contained in a member variable named **m_localTime**, of type **sc_core::sc_time**. The local time can be accessed with the following accessors: **addLocalTime()**, **setLocalTime()** and **getLocalTime()**.

```
sc_core::sc_time m_localTime;                    // the initiator local time
...
void addLocalTime(sc_core::sc_time t);           // add a value to the local time
void setLocalTime(sc_core::sc_time& t);          // set the local time
sc_core::sc_time getLocalTime(void);             // get the local time
```

The initiator activity corresponds to the boolean member **m_activity** that indicates if the initiator is currently active (i.e. **true**, wants to participate in the arbitration in the interconnect) or inactive (i.e. **false**, does not want to participate in the arbitration in the interconnect). The corresponding access functions are **setActivity()** and **getActivity()**.

```
bool m_activity;
...
void setActivity(bool t);                        // set the activity status (true if the comp
bool getActivity(void);                          // get the activity state
```

The **execLoop()** method, describing the initiator behaviour must be declared as a member function of the **my_initiator** class.

Finally, the class **my_initiator** must contain a member variable **p_vci_init**, of type **tlmt_simple_initiator_socket**. This member variable represents the VCI initiator port. It has 3 template parameters, two of which are used to help connecting the response callback function (**my_initiator** in the example, first template parameter) to the port and defining the port type (**soclib_vci_types** in the following example, third template parameter). **soclib_vci_types** is indeed a C++ structure containing two typedef: the first typedef defines the payload type as VCI, and the other defines the TLM phase type. The phase type can either be **TLMT_CMD** (i.e. the transaction indicates the emission of a command by an initiator and its reception by a target), **TLMT_RSP** (i.e. the transaction indicates the emission of a response by a target and its reception by an initiator), or **TLMT_INFO** (i.e. a TLM-T transaction emitted by one side of a link (vci, irq or fifo) to get information such as time and activity on the other side of the link).

C.1) Sending a VCI command packet

To send a VCI command packet, the **execLoop()** method must use the **nb_transport_fw()** method, that is a member function of the **p_vci_init** port. The prototype of this method is the following:

```
tlm::tlm_sync_enum nb_transport_fw              /// sync status
( soclib_vci_types::tlm_payload_type &payload,      ///< VCI payload pointer
  soclib_vci_types::tlm_phase_type   &phase,       ///< transaction phase
  sc_core::sc_time                   &time);        ///< time
```

The first parameter of this member function is the VCI packet, the second represents the phase (TLMT_CMD in this case), and the third parameter contains the initiator local time.

To prepare a VCI packet for sending, the **execLoop** function must declare two objects locally, **payload** and **phase**.

```
soclib_vci_types::tlm_payload_type payload;
soclib_vci_types::tlm_phase_type phase;
```

A payload of type **soclib_vci_types::tlm_payload_type** corresponds to a **tlmt_vci_transaction**. It contains three groups of information:

- TLM2.0 related fields
- TLM-T related fields
- VCI related fields

The contents of a **tlmt_vci_transaction** is defined below:

```
class tlmt_vci_transaction
{
        ...
private:
  // TLM2.0 related fields and common structure

  sc_dt::uint64        m_address;              // address
  unsigned char*       m_data;                 // buf
  unsigned int         m_length;               // nword
  tlmt_response_status m_response_status;      // rerror
  bool                 m_dmi;                  // nothing
  unsigned char*       m_byte_enable;          // be
  unsigned int         m_byte_enable_length;
  unsigned int         m_streaming_width;      //

  // TLM-T related fields

  bool*                m_activity_ptr;
  sc_core::sc_time*    m_local_time_ptr;

  // VCI related fields

  tlmt_command         m_command;              // cmd
  unsigned int         m_src_id;               // srcid
  unsigned int         m_trd_id;               // trdid
  unsigned int         m_pkt_id;               // pktid
```

The TLM2.0 compliant accessors allow to set the TLM2.0 related fields, such as the transaction address, the byte enable array pointer and its associated size in bytes, and the data array pointer and its associated size in bytes. The byte enable array allows to build versatile packets thanks to a powerful but slow data masking scheme. Further experiments are currently done to evaluate the performance degradation incurred by the byte formatting. It is therefore possible that the types of the **m_data** and **m_byte_enable** of the **tlmt_vci_transaction** will be changed to **uint32*** in a near future.

Dedicated VCI accessors are used to define the VCI transaction type, that can either be **set_read()** (for read command), **set_write()** (for write command),**set_locked_read()** (for atomic locked read), and **set_store_cond()** (for atomic store conditional). The **set_src_id()**, **set_trd_id()** and **set_pkt_id()** functions respectively set the VCI source, thread and packet identifiers. The following example describes a VCI write command:

```
payload.set_address(0x10000000);//ram 0
payload.set_byte_enable_ptr(byte_enable);
payload.set_byte_enable_length(nbytes);
payload.set_data_ptr(data);
payload.set_data_length(nbytes); // 5 words of 32 bits

payload.set_write();
payload.set_src_id(m_id);
payload.set_trd_id(0);
payload.set_pkt_id(pktid);

phase= soclib::tlmt::TLMT_CMD;
sendTime = getLocalTime();
```

```
p_vci_init->nb_transport_fw(payload, phase, sendTime);
```

The **nb_transport_fw()** function is non-blocking. To implement a blocking transaction (such as a cache line read, where the processor is stalled during the VCI transaction), the model designer must use the SystemC **sc_core::wait(x)** primitive (**x** being of type **sc_core::sc_event**): the **execLoop()** thread is then suspended, and will be reactivated when the response packet is actually received.

C.2) Receiving a VCI response packet

To receive a VCI response packet, a call-back function must be defined as a member function of the class **my_initiator**. This call-back function (named **vci_rsp_received()** in the example), must be declared in the **my_initiator** class and is executed each time a VCI response packet is received on the **p_vci_init** port. The function name is not constrained, but the arguments must respect the following prototype:

```
tlm::tlm_sync_enum vci_rsp_received
  ( soclib_vci_types::tlm_payload_type &payload,      // payload
    soclib_vci_types::tlm_phase_type   &phase,       // transaction phase
    sc_core::sc_time                   &time);        // resp time
```

The return value (type tlm::tlm_sync_enum) must be sytematically set to tlm::TLM_COMPLETED in this implementation The function parameters are identical to those described in the forward transport function

In the general case, the actions executed by the call-back function depend on the response transaction type (**m_command** field), as well as the **pktid** and **trdid** fields. For sake of simplicity, the call-back function proposed below does not make any distinction between VCI transaction types.

C.3) Initiator Constructor

The constructor of the class **my_initiator** must initialize all the member variables, including the **p_vci_init** port. The **vci_rsp_received()** call-back function being executed in the context of the thread sending the response packet, a link between the **p_vci_init** port and this call-back function must be established.

The **my_initiator** constructor for the **p_vci_init** object must be called with the following arguments:

```
p_vci_init.register_nb_transport_bw(this, &my_initiator::vci_rsp_received);
```

C.4) Lookahead parameter

The SystemC simulation engine behaves as a cooperative, non-preemptive multi-tasks system. Any thread in the system must stop execution after at some point, in order to allow the other threads to execute. With the proposed approach, a TLM-T initiator will never stop if it does not execute blocking communication (such as a processor that has all code and data in the L1 caches).

To solve this issue, it is necessary to define -for each initiator module- a **lookahead** parameter. This parameter defines the maximum number of cycles that can be executed by the thread before it is automatically stopped. The **lookahead** parameter allows the system designer to bound the de-synchronization time interval between threads.

A small value for this parameter results in a better timing accuracy for the simulation, but implies a larger number of context switches, and a slower simulation speed.

C.4) VCI initiator example

```
////////////////////////// my_initiator.h ////////////////////////////////

////////////////////////// my_initiator.cpp ////////////////////////////////
```

D) VCI target modeling

In the proposed example, the **my_target** component handles all VCI commands in the same way, and there is no error management.

The class **my_target** inherits from the class **sc_core::sc_module**. The class **my_target** contains a member variable **p_vci_target** of type **tlmt_simple_target_socket**. This object has 3 template parameters, that are identical to those used for declaring initiator ports (see above).

D.1) Receiving a VCI command packet

To receive a VCI command packet, a call-back function must be defined as a member function of the class **my_target**. This call-back function (named **vci_cmd_received()** in the example), will be executed each time a VCI command packet is received on the **p_vci_target** port. The function name is not constrained, but the arguments must respect the following prototype:

```
tlm::tlm_sync_enum vci_cmd_received
( soclib_vci_types::tlm_payload_type &payload,      // VCI payload pointer
  soclib_vci_types::tlm_phase_type   &phase,       // transaction phase
  sc_core::sc_time                   &time);        // time
```

D.2) Sending a VCI response packet

To send a VCI response packet the call-back function **vci_cmd_received()** must use the **nb_transport_bw()** method, that is a member function of the class **tlmt_simple_target_socket**, and has the same arguments as the **nb_transport_fw()** function. Respecting the general TLM2.0 policy, the payload argument refers to the same **tlmt_vci_transaction** object for both the **nb_transport_fw()** and **nb_transport_bw()** functions, and the associated call-back functions. The set_response_status field must be documented for all transaction types, but only two values are used in this implementation:

- TLMT_OK_RESPONSE
- TLMT_ERROR_RESPONSE

For a reactive target, the response packet time is computed as the command packet time plus the target intrinsic latency.

```
payload.set_response_status(soclib::tlmt::TLMT_OK_RESPONSE);
phase = soclib::tlmt::TLMT_VCI_RSP;
time = time + (nwords * UNIT_TIME);
p_vci_target->nb_transport_bw(payload, phase, time);
```

D.3) Target Constructor

The constructor of the class **my_target** must initialize all the member variables, including the **p_vci_target** port. The **vci_cmd_received()** call-back function being executed in the context of the thread sending the command packet, a link between the **p_vci_target** port and the call-back function must be established. The **my_target** constructor must be called with the following arguments:

```
p_vci_target.register_nb_transport_fw(this, &my_target::vci_cmd_received);
```

D.4) VCI target example

```
////////////////////////// my_target.h ////////////////////////////////
```

```
////////////////////////// my_target.cpp ///////////////////////////////
```

E) VCI Interconnect modelling

The VCI interconnect used for the TLM-T simulation is a generic simulation model, named **VciVgmn**. The two main parameters are the number of initiators, and the number of targets. In TLM-T simulation, we don't want to reproduce the cycle-accurate behavior of a particular interconnect. We only want to simulate the contention in the network, when several VCI intitiators try to reach the same VCI target. Therefore, the network is actually modeled as a complete cross-bar : In a physical network such as the multi-stage network described in Figure 2.a, conflicts can appear at any intermediate switch. In the **VciVgmn** network described in Figure 2.b, conflicts can only happen at the output ports. It is possible to specify a specific latency for each input/output couple. As in most physical interconnects, the general arbitration policy is round-robin.

E.1) Generic network modeling

There is actually two fully independent networks for VCI command packets and VCI response packets. There is a routing function for each input port, and an arbitration function for each output port, but the two networks are not symmetrical :

- For the command network, the arbitration policy is distributed: there is one arbitration thread for each output port (i.e. one arbitration thread for each VCI target). Each arbitration thread is modeled by a SC_THREAD, and contains a local clock.
- For the response network, there are no conflicts, and there is no need for arbitration. Therefore, there is no thread (and no local time) and the response network is implemented by simple function calls.

This is illustrated in Figure 3 for a network with 2 initiators and three targets :

E.2) Arbitration Policy

As described above, there is one arbitration thread associated to each VCI target. This thread

E.3) General Initiators and Targets synchronization

As described in sections B & C, each VCI initiator TLM-T module contains a thread and a local clock. But, in order to increase the TLM-T simulation speed, the VCI targets are generally described as reactive call-back functions. Therefore, there is no thread, and no local clock in the TLM-T module describing a VCI target. For a VCI target, the local clock is actually the clock associated to the corresponding arbitration thread contained in the **VciVgmn** module.