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

Authors : Alain Greiner, François Pêcheux, Aline Vieira de Mello

1. A) Introduction
2. B) VCI initiator and VCI target
3. C) VCI Transaction in TLM-T
4. D) VCI initiator Modeling
    1. D.1) Member variables & methods
    2. D.2) Sending a VCI command packet
    3. D.3) Receiving a VCI response packet
    4. D.4) Initiator Constructor
    5. D.5) Local Time Representation & Synchronization
    6. D.6) VCI initiator example
5. E) VCI target modeling
    1. E.1) Member variables & methods
    2. E.2) Receiving a VCI command packet
    3. E.3) Sending a VCI response packet
    4. E.4) Target Constructor
    5. E.5) VCI target example
6. F) VCI Interconnect modeling
    1. F.1) Generic network modeling

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. In the TLM-T approach, we don't use the SystemC global time, as each PDES process involved in the simulation has its own local time. PDES processes (implemented as SC_THREADS) synchronize through messages piggybacked with time information. This timing information is actually the absolute locat time of the sender. Models complying to these TLM-T rules can be used with the "standard" OSCI simulation engine (SystemC 2.x) and the TLM2.0 library, but can also be used also with others simulation engines, especially parallelized simulation engines.

The pessimistic PDES algorithm relies on temporal filtering of the incoming messages. A PDES process that has N input channels is only allowed to process when it has timing information on all its input ports. For example, an interconnect is only allowed to let a command packet reach a given target when all the initiators that can address this target have sent at least one timed message. To solve this issue the PDES algorithm uses *nul message*. A null message contains no data, but only a time information. Moreover, all processes can be in two modes : active & non-active. Only processes that are active participate to the temporal filtering.

A first implementation of TLMT used *sollicited null messages*, but the final solution uses *direct null-messages*, that strictly follow the Chandy-Misra pessimistic algorithm. Each process cannot run independently without sending a timed message for a time larger that a predefined value, called the SYNCHRONIZATION_TIME_QUANTUM. When this time quantum is elapsed, the process must send a null message on its output ports.

The models following the writing rules defined herein are syntactically compliant with the TLM2.0 standard, but have a different representation for the time. In particular, the third parameter of the transport functions is considered to be an absolute (but local) time and is not an offset relative to a global simulation time that is not used

anymore. The examples presented below use the VCI/OCP communication protocol selected by the SoCLib project, but the TLM-T approach described here is very flexible, and is not limited to the VCI/OCP communication protocol.

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

B) VCI initiator and VCI target

Figure 1 presents a minimal system containing one single VCI initiator, **my_initiator** , and one single VCI target, **my_target** . The initiator behavior is modeled by the SC_THREAD **execLoop()**, that contains an infinite loop. The interface function **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 interface function. In other words, there is no need to use a SC_THREAD for a target component: the target behaviour is entirely described by the interface function **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-directional channel, encapsulating two separated uni-directional channels: one to transmit the VCI command packet in the **nb_transport_fw()** function, one to transmit the VCI response packet in the **nb_transport_bw()** function.

C) VCI Transaction in TLM-T

The TLM2.0 standard defines a generic payload that contains almost all the fields needed to implement the complete vci protocol. In SocLib, the missing fields are defined in what TLM2.0 calls a payload extension. The C++ class used to implement this extension is **soclib_payload_extension**.

The SocLib payload extension only contains four data members:

```
command                 m_soclib_command;
unsigned int            m_src_id;
unsigned int            m_trd_id;
unsigned int            m_pkt_id;
```

The **m_soclib_command** data member supersedes the command of the TLM2.0 generic payload. The parameter to the **set_command()** of a generic payload is always set to **tlm::TLM_IGNORE_COMMAND**. Seven values can be assigned to **m_soclib_command**. These values are:

```
VCI_READ_COMMAND,
VCI_WRITE_COMMAND,
VCI_LINKED_READ_COMMAND,
VCI_STORE_COND_COMMAND,
PDES_NULL_MESSAGE,
PDES_ACTIVE,
PDES_INACTIVE
```

The **VCI_READ_COMMAND** (resp. **VCI_WRITE_COMMAND**) is used to send a VCI read (resp. write) packet command. The **VCI_LINKED_READ_COMMAND** and **VCI_STORE_CONDITIONAL_COMMAND** are used to implement atomic operations. The latter 3 values are not directly related to VCI but rather to the PDES simulation algorithm. The **PDES_NULL_MESSAGE** value is used whenever an initiator needs to send its local time to the rest of the platform for synchronization purpose. The **PDES_ACTIVE** and **PDES_INACTIVE** values are used to inform the interconnect that the corresponding initiator must be taken into account in the temporal

filtering or not. For example, a programmable DMA controller should not participate in the PDES temporal filtering until it has been programmed and launched. At the beginning of the simulation, all the initiators send at least one synchronization message.

The data members of the **soclib_payload_extension** can be accessed through the following access functions:

```
// Command related methods
bool         is_read() const {return (m_soclib_command == VCI_READ_COMMAND);}
void         set_read() {m_soclib_command = VCI_READ_COMMAND;}
bool         is_write() const {return (m_soclib_command == VCI_WRITE_COMMAND);}
void         set_write() {m_soclib_command = VCI_WRITE_COMMAND;}
bool         is_locked_read() const {return (m_soclib_command == VCI_LINKED_READ_COMMAND);}
void         set_locked_read() {m_soclib_command = VCI_LINKED_READ_COMMAND;}
bool         is_store_cond() const {return (m_soclib_command == VCI_STORE_COND_COMMAND);}
void         set_store_cond() {m_soclib_command = VCI_STORE_COND_COMMAND;}
bool         is_null_message() const {return (m_soclib_command == PDES_NULL_MESSAGE);}
void         set_null_message() {m_soclib_command = PDES_NULL_MESSAGE;}
bool         is_active() const {return (m_soclib_command == PDES_ACTIVE);}
void         set_active() {m_soclib_command = PDES_ACTIVE;}
bool         is_inactive() const {return (m_soclib_command == PDES_INACTIVE);}
void         set_inactive() {m_soclib_command = PDES_INACTIVE;}
enum command get_command() const {return m_soclib_command;}
void         set_command(const enum command c) {m_soclib_command = c;}

// identification related methods
unsigned int get_src_id() const {return m_src_id;}
void         set_src_id(unsigned int id) {m_src_id = id;}
unsigned int get_trd_id() const {return m_trd_id;}
void         set_trd_id(unsigned int id) {m_trd_id = id;}
unsigned int get_pkt_id() const {return m_pkt_id;}
void         set_pkt_id(unsigned int id) {m_pkt_id = id;}
```

To build a new VCI packet, one has to create a generic payload and a soclib payload extension, and to call the appropriate access functions on these two objects. For example, to issue a VCI read command, one should write the following code:

```
tlm::tlm_generic_payload *payload_ptr = new tlm::tlm_generic_payload();
soclib_payload_extension *extension_ptr = new soclib_payload_extension();
...
// set the values in tlm payload
payload_ptr->set_command(tlm::TLM_IGNORE_COMMAND);
payload_ptr->set_address(0x10000000]);
payload_ptr->set_byte_enable_ptr(byte_enable);
payload_ptr->set_byte_enable_length(nbytes);
payload_ptr->set_data_ptr(data);
payload_ptr->set_data_length(nbytes);
// set the values in payload extension
extension_ptr->set_read();
extension_ptr->set_src_id(m_srcid);
extension_ptr->set_trd_id(0);
extension_ptr->set_pkt_id(pktid);
// set the extension to tlm payload
payload_ptr->set_extension (extension_ptr);
...
```

D) VCI initiator Modeling

D.1) Member variables & methods

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 uses the class **pdes_local_time** for managing and interacting with his local time and with the interval between two consecutive null messages. The **pdes_local_time** has the following access functions:

```
sc_core::sc_time m_local_time;                      // the initiator local time
sc_core::sc_time m_next_sync_point;                 // the next synchronization point
sc_core::sc_time m_time_quantum;                    // the time quantum
...
pdes_local_time(sc_core::sc_time time_quantum);     // constructor
void add(const sc_core::sc_time& t);                // add an increment to the local time
void set(sc_core::sc_time t);                       // set the local time
sc_core::sc_time get();                             // get the local time
bool need_sync();                                   // check if a synchronization is required
```

The initiator activity status (used by the temporal filtering, as described in section F) is managed for the class **pdes_activity_status**. The corresponding access functions are **set()** and **get()**.

```
bool m_activity_status;                          // the initiator activity status
...
pdes_activity_status();                          // constructor
void set(bool a);                                // set the activity status (true if the comp
bool get();                                      // get the activity state
```

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

The **my_initiator** class contains a member variable **p_vci_init**, of type **tlm_utils::simple_initiator_socket**, representing the VCI initiator port.

It must also define an interface function to handle the VCI response packets.

D.2) Sending a VCI command packet

To send a VCI command packet, the **execLoop()** method must use the **nb_transport_fw()** method, defined by TLM2.0, 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
( tlm::tlm_generic_payload &payload,      // payload
  tlm::tlm_phase           &phase,        // phase (TLM::BEGIN_REQ)
  sc_core::sc_time         &time);        // absolute local time
```

The first argument is a pointer to the payload (including the soclib payload extension), the second represents the phase (always set to TLM::BEGIN_REQ for requests), and the third argument contains the initiator local time. The return value is not used in this TLM-T implementation.

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.

D.3) Receiving a VCI response packet

To receive a VCI response packet, an interface function must be defined as a member function of the class **my_initiator**. This function (named **nb_transport_bw()** in the example), must be linked to the **p_vci_init** port, 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 nb_transport_bw
( tlm::tlm_generic_payload &payload,      // payload
  tlm::tlm_phase           &phase,        // phase (TLM::BEGIN_RESP)
  sc_core::sc_time         &time);        // response time
```

The return value (type tlm::tlm_sync_enum) is not used in this TLM-T implementation, and must be sytematically set to tlm::TLM_COMPLETED.

D.4) Initiator Constructor

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

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

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

D.5) Local Time Representation & Synchronization

The SystemC simulation engine behaves as a cooperative, non-preemptive multi-tasks system. Any thread in the system must stop execution at some point, in order to allow the other threads to execute. Moreover each PDES process must send null message periodically.

To solve this issue, it is necessary to define -for each initiator module- a **synchronization time quantum** parameter. This parameter defines the maximum delay between two successive timed messages. When this time quantum is elapsed, the component send a null message, and the corresponding thread is descheduled.

This time quantum mechanism is implemented in the **pdes_local_time** class. For each initiator, the time quantum value is a parameter defined as a constructor argument. The three members methods are...

D.6) VCI initiator example

```
#include "my_initiator.h"                // header

my_initiator::my_initiator
( sc_core::sc_module_name name,          // module name
  const soclib::common::IntTab &index,   // index of mapping table
  const soclib::common::MappingTable &mt, // mapping table
  sc_core::sc_time time_quantum)         // time quantum
  : sc_module(name),                     // init module name
    m_mt(mt),                            // mapping table
    p_vci_init("socket")                 // vci initiator socket name
{
  //register callback function (VCI INITIATOR SOCKET)
  p_vci_init.register_nb_transport_bw(this, &my_initiator::my_nb_transport_bw);
   //initiator identification
  m_srcid = mt.indexForId(index);
  //PDES local time
  m_pdes_local_time = new pdes_local_time(time_quantum);
  //PDES activity status
  m_pdes_activity_status = new pdes_activity_status();
  // register thread process
  SC_THREAD(execLoop);
}
```

```
// send to interconnect the initiator activity status
void my_initiator::sendActivity()
{
  tlm::tlm_generic_payload *payload_ptr = new tlm::tlm_generic_payload();
  soclib_payload_extension *extension_ptr = new soclib_payload_extension();
  tlm::tlm_phase            phase;
  sc_core::sc_time          time;

  // set the active or inactive command
  if(m_pdes_activity_status->get()) extension_ptr->set_active();
  else extension_ptr->set_inactive();
  // set the extension to tlm payload
  payload_ptr->set_extension (extension_ptr);
  //set the tlm phase
  phase = tlm::BEGIN_REQ;
  //set the local time to transaction time
  time = m_pdes_local_time->get();
  //send a message with command equals to PDES_ACTIVE or PDES_INACTIVE
  p_vci_init->nb_transport_fw(*payload_ptr, phase, time);
  //wait a response
  wait(m_rspEvent);
}

// send to interconnect a null message with the initiator local time
void my_initiator::sendNullMessage()
{
  tlm::tlm_generic_payload *payload_ptr = new tlm::tlm_generic_payload();
  soclib_payload_extension *extension_ptr = new soclib_payload_extension();
  tlm::tlm_phase            phase;
  sc_core::sc_time          time;

  // set the null message command
  extension_ptr->set_null_message();
  // set the extension to tlm payload
  payload_ptr->set_extension(extension_ptr);
  //set the tlm phase
  phase = tlm::BEGIN_REQ;
  //set the local time to transaction time
  time = m_pdes_local_time->get();
  //send a null message
  p_vci_init->nb_transport_fw(*payload_ptr, phase, time);
  //deschedule the initiator thread
  wait(sc_core::SC_ZERO_TIME);
}

// initiator thread
void my_initiator::execLoop(void)
{
  tlm::tlm_generic_payload *payload_ptr = new tlm::tlm_generic_payload();
  soclib_payload_extension *extension_ptr = new soclib_payload_extension();
  tlm::tlm_phase phase;
  sc_core::sc_time time;
  uint32_t nbytes = 4;
  unsigned char data[nbytes];
  unsigned char byte_enable[nbytes];

  while (true){
    //fill the byte_enable and data
    for(unsigned int i=0; i<nbytes; i++){
      byte_enable[i]=0xFF;
      data[i]=0xAA;
    }

    // set the values in tlm payload
    payload_ptr->set_command(tlm::TLM_IGNORE_COMMAND);
```

```
    payload_ptr->set_address(0x10000000);
    payload_ptr->set_byte_enable_ptr(byte_enable);
    payload_ptr->set_byte_enable_length(nbytes);
    payload_ptr->set_data_ptr(data);
    payload_ptr->set_data_length(nbytes);
    // set the values in payload extension
    extension_ptr->set_write();
    extension_ptr->set_src_id(m_srcid);
    extension_ptr->set_trd_id(0);
    extension_ptr->set_pkt_id(0);
    // set the extension to tlm payload
    payload_ptr->set_extension (extension_ptr );
    // set the tlm phase
    phase = tlm::BEGIN_REQ;
    // set the local time to transaction time
    time = m_pdes_local_time->get();
    // send the transaction and wait a response
    p_vci_init->nb_transport_fw(*payload_ptr, phase, time);
    wait(m_rspEvent);

    // increment the local time
    m_pdes_local_time->add(10 * UNIT_TIME);
    // if a synchronization is nexcessary then the initiator sends a null message
    if (m_pdes_local_time->need_sync()) {
      sendNullMessage();
    }
  } // end while true

  //desactive the initiator and inform to interconnect
  m_pdes_activity_status->set(false);
  sendActivity();
}

// inbound nb_transport_bw (VCI INITIATOR SOCKET)
tlm::tlm_sync_enum my_initiator::my_nb_transport_bw
( tlm::tlm_base_protocol_types::tlm_payload_type &payload,
  tlm::tlm_base_protocol_types::tlm_phase_type   &phase,
  sc_core::sc_time                               &time)
{
  // update the local time
  m_pdes_local_time->set(time);
  // wake up the initiator thread
  m_rspEvent.notify(sc_core::SC_ZERO_TIME);
  return tlm::TLM_COMPLETED;
}
```

E) VCI target modeling

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

E.1) Member variables & methods

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 **tlm_utils::simple_target_socket**, representing the VCI target port. It contains an interface function to handle the received VCI command packets, as described below.

E.2) Receiving a VCI command packet

To receive a VCI command packet, an interface function must be defined as a member function of the class **my_target**. This function (named **nb_transport_fw()** in the example), is 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 nb_transport_fw
( tlm::tlm_generic_payload &payload,      // payload
  tlm::tlm_phase           &phase,        // phase (TLM::BEGIN_REQ)
  sc_core::sc_time         &time);        // time
```

The return value (type tlm::tlm_sync_enum) is not used in this TLM-T implementation, and must be sytematically set to tlm::TLM_COMPLETED.

E.3) Sending a VCI response packet

To send a VCI response packet the call-back function uses the **nb_transport_bw()** and has the same arguments as the **nb_transport_fw()** function. Respecting the general TLM2.0 policy, the payload argument refers to the same **tlm_generic_payload** object for both the **nb_transport_fw()** and **nb_transport_bw()** functions, and the associated interface functions. Only two values are used for the **response_status** field in this TLM-T implementation:

- TLM_OK_RESPONSE
- TLM_GENERIC_ERROR_RESPONSE

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

```
tlm::tlm_sync_enum nb_transport_bw
( tlm::tlm_generic_payload &payload,
  tlm::tlm_phase           &phase,
  sc_core::sc_time         &time)
{
  ...
  payload.set_response_status(tlm::TLM_OK_RESPONSE);
  phase = tlm::BEGIN_RESP;
  time = time + (nwords * UNIT_TIME);
  p_vci_target->nb_transport_bw(payload, phase, time);
}
```

E.4) Target Constructor

The constructor of the class **my_target** must initialize all the member variables, including the **p_vci_target** port. The **nb_transport_fw()** 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::nb_transport_fw);
```

E.5) VCI target example

```
#include "my_target.h"                              // header

my_target::my_target
```

```
( sc_core::sc_module_name name,                       // module name
  const soclib::common::IntTab &index,                // index of mapping table
  const soclib::common::MappingTable &mt)             // mapping table
  : sc_module(name),                                  // init module name
    m_mt(mt),                                         // mapping table
    p_vci_target("socket")                            // vci target socket name
{
  // register callback fuction (VCI TARGET SOCKET)
  p_vci_target.register_nb_transport_fw(this, &my_target::my_nb_transport_fw);
  // identification
  m_tgtid = m_mt.indexForId(index);
}

// inbound nb_transport_fw (VCI TARGET SOCKET)
tlm::tlm_sync_enum my_target::my_nb_transport_fw
( tlm::tlm_generic_payload &payload,
  tlm::tlm_phase           &phase,
  sc_core::sc_time         &time)
{
  // get the payload extension
  soclib_payload_extension *extension_pointer;
  payload.get_extension(extension_pointer);
  // get the number of words
  uint32_t nwords = payload.get_data_length() / vci_param::nbytes;

  switch(extension_pointer->get_command()){
  case soclib::tlmt::VCI_READ_COMMAND:
  case soclib::tlmt::VCI_WRITE_COMMAND:
  case soclib::tlmt::VCI_LINKED_READ_COMMAND:
  case soclib::tlmt::VCI_STORE_COND_COMMAND:
    ...

    //set ok to response status
    payload.set_response_status(tlm::TLM_OK_RESPONSE);
    break;
  default:
    //set error to response status
    payload.set_response_status(tlm::TLM_GENERIC_ERROR_RESPONSE);
    break;
  }

  //modify the phase
  phase = tlm::BEGIN_RESP;
  //increment the target processing time
  time = time + (nwords * UNIT_TIME);
  //send the response
  p_vci_target->nb_transport_bw(payload, phase, time);
  return tlm::TLM_COMPLETED;
}
```

F) VCI Interconnect modeling

The VCI interconnect used for the TLM-T simulation is a generic interconnection network, 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 detailed, 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.

In a physical network such as the multi-stage network described in Figure 2.a, conflicts can appear at any intermediate switch. The **VciVgmn** network, described in Figure 2.b, is modeled as a cross-bar, and 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.

F.1) Generic network modeling

According to PDES, a packet P emitted by an initiator reaches the correct target when it is safe to do so, i.e. when the interconnect is sure that no initiator will send a packet with a timestamp lesser than the timestamp of P. This temporal filtering operation can be factorized, when all the connected active initiators have sent at least one message to the interconnect. These messages are stored in a centralized data structure. This structure stores tree information: the packet, the timestamps and the current initiator activity. After elaboration of the simulator, the activity information for each initiator is set to true. A coprocessor initiator will send a message with **m_soclib_command** set to **TLMT_INACTIVE** at the beginning of the simulation. Therefore, when all slots of this centralized structure are filled with real or null messages with their associated timestamps, a temporal filtering iteration can occur.

The arbitration process must take into account the actual state of the VCI initiators: For example a DMA coprocessor that has not yet been activated will not send request and should not participate in the temporal filtering and arbitration process. As a general rule, each VCI initiator must define an **active** boolean flag, defining if it should participate to the arbitration. This **active** flag is always set to true for general purpose processors.

There are actually two fully independent networks for VCI command packets and VCI response packets.

The two networks are not symmetrical :

- There is one processing thread for each output port (i.e. one processing thread for each VCI target). Each processing thread is modeled by a SC_THREAD, and contains a dedicated message fifo and a local time. This time represents the target local time.
- For the response network, there are no conflicts, and therefore there is no thread (and no local time). The response network is implemented by simple function calls.

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

The command network handles the two following tasks:

- Temporal filtering and arbitration of the requests from the initiators.

This task is activated when all the connected initiators have sent at least one message to the interconnect. The task computes the list of the messages that can actually be sent to the targets according to PDES. The list contains all the messages which timestamp belongs to the time interval [T, T+ interconnect_delay], where T is the smallest timestamp of all the messages in the interconnect. Priority between initiators with the same local time is computed using a traditional round-robin algorithm. The temporal filtering and arbitration task is executed in the context of the initiator that sends a (possibly null) message.

- Routing of a filtered request packet to the correct target. Each target runs under the control of a processing thread and

has a dedicated message fifo. The routing wakes up the processing thread of the corresponding target, that empties the message fifo filled by the temporal filtering. The behavioral function of the target is executed in the context of the processing thread.