# Writing efficient TLM-T SystemC simulation models for SoCLib

Authors : Alain Greiner, François Pêcheux, Emmanuel Viaud, Nicolas Pouillon

1. A) Introduction
2. B) Single VCI initiator and single VCI target
3. C) VCI initiator Modeling
    1. C.1) Sending a VCI command packet
    2. C.2) Receiving a VCI response packet
    3. C.3) Initiator Constructor
    4. C.4) Lookahead parameter
    5. C.4) VCI initiator example
4. D) VCI target modeling
    1. D.1) Receiving a VCI command packet
    2. D.2) Sending a VCI response packet
    3. D.3) Target Constructor
    4. D.4) VCI target example
5. E) VCI Interconnect
    1. E.1) Generic network modeling
    2. E.2) VCI initiators and targets synchronisations
6. F) Interrupt modeling
    1. F.1) Source modeling
    2. F.2) Destination modeling
    3. F.3) Processor with interrupt example

# A) Introduction

Important note: this document is currently being updated and deeply modified. A stable version will be available on February 22, 2008.

This document describes the modeling rules for writing TLM-T SystemC simulation models for SoCLib. 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), 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 system containing one single initiator, **VciSimpleInitiator** , and one single target, **VciSimpleTarget** . The **VciSimpleInitiator** module behavior is modeled by the SC_THREAD **execLoop()**, that contains an infinite loop. The call-back function **rspReceived()** 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 call-back function. Actually, there is no need to use a SC_THREAD : The target behaviour is entirely described by the call-back function **cmdReceived()**, 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 **VciSimpleInitiator** class. This class inherits from the **soclib::tlmt::BaseModule** class, that acts as the root class for all TLM-T modules. The process time, as well as other useful process attributes, is contained in a C++ structure called a **tlmt_core::tlmt_thread_context**.As there is only one thread in this module, there is only one member variable **c0** of type **tlmt_core::tlmt_thread_context**. **c0** mostly corresponds to the PDES process local time (**H** on the figure). In all the TLM-T models, time is handled by the **tlmt_core::tlmt_time** class. In order to get the time associated to the initiator process, one should use the getter function **time()** on the **c0** object, and the setter functions **set_time()** and **update_time()** to assign a new time to the process.

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

Finally, the class **VciSimpleInitiator** must contain a member variable **p_vci**, of type **VciInitiator**. This object has a template parameter **<vci_param>** defining the widths of the VCI ADDRESS and DATA fields.

## C.1) Sending a VCI command packet

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

```
void send(vci_cmd_packet<vci_param> *cmd,       // VCI command packet
          tlmt_core::tlmt_time time);           // initiator local time
```

The informations transported by a VCI command packet are defined below:

```
template<typename vci_param>
class vci_cmd_packet
{
public:
        typename vci_param::cmd_t cmd;        // VCI transaction type
        typename vci_param::addr_t *address;  // pointer to an array of addresses on the target
        unsigned int be;                      // byte_enable value (same value for all packet wo
        bool contig;                          // contiguous addresses (when true)
        typename vci_param::data_t *buf;      // pointer to the local buffer on the initiator
        size_t length;                        // number of words in the packet
        uint32_t srcid;                       // VCI Source ID
        uint32_t trdid;                       // VCI Thread ID
        uint32_t pktid;                       // VCI Packet ID
};
```

The possible values for the **cmd** filed are `vci_param::CMD_READ`, `vci_param::CMD_WRITE`, `vci_param::CMD_READ_LOCKED`, and `vci_param::CMD_STORE_COND`. The **contig** field can be used for optimisation.

The **send()** function is non-blocking. To implement a blocking transaction (such as a cache line read, where the processor is *frozen* 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 **VciSimpleInitiator**. This call-back function (named **rspReceived()** in the example), is executed each time a VCI response packet is received on the **p_vci** port. The function name is not constrained, but the arguments must respect the following prototype:

```
tmpl(tlmt_core::tlmt_return&)::callback(soclib::tlmt::vci_rsp_packet<vci_param> *pkt,
            const tlmt_core::tlmt_time &time,
            void *private_data)
```

The informations transported by a VCI response packet are defined below:

```
template<typename vci_param>
class vci_rsp_packet
{
public:
        typename vci_param::cmd_t cmd;    // VCI transaction type
        size_t length;                    // number of words in the packet
        uint32_t error;                   // error code (0 if no error)
        uint32_t srcid;                   // VCI Source ID
        uint32_t trdid;                   // VCI Thread ID
        uint32_t pktid;                   // VCI Packet ID
};
```

The actions executed by the call-back function depend on the response transaction type (**cmd** field), as well as the **pktid** and **trdid** fields.

In the proposed example :

- In case of of a blocking read , the call-back function updates the local time, and activates the suspended thread.
- In case of a non-blocking write, the call-back function does nothing.

## C.3) Initiator Constructor

The constructor of the class **VciSimpleInitiator** must initialize all the member variables, including the **p_vci** port. The **rspReceived()** call-back function being executed in the context of the thread sending the response packet, a link between the **p_vci** port and the call-back function must be established. Moreover, the **p_vci** port must contain a pointer to the initiator thread context **c0**.

The **VciInitiator** constructor for the **p_vci_** object must be called with the following arguments:

```
p_vci("vci", new tlmt_core::tlmt_callback<VciSimpleInitiator,soclib::tlmt::vci_rsp_packet<vci_pa
                                         this, &VciSimpleInitiator<vci_param>::rspReceived), &c0
```

## 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

```
template <typename vci_param>
class my_initiator : tlmt::BaseModule {
public:
    VciInitiatorPort <vci_param>  p_vci;

    //////// constructor
    my_initiator (sc_module_name name,
                         uint32_t initiatorIndex
                         uint32_t lookahead) :
    p_vci(?vci?, this, &my_initiator::rspReceived, &m_time),
    tlmt::BaseModule(name),
    m_time(0)
    {
    m_index = InitiatorIndex;
    m_lookahed = lookahead;
    m_counter = 0;
    SC_THREAD(execLoop);
    } // end constructor

private:
    tlmt_time m_time;  // local clock
    uint32_t m_index;  // initiator index
    uint32_t m_counter;  // iteration counter
    uint32_t m_lookahed;  // lookahead value
    vci_param::data_t m_data[8];         // local buffer
    vci_cmd_t   m_cmd;  // paquet VCI commande
    sc_event  m_rsp_received; // synchronisation signal

    //////// thread
    void execLoop()
    {
    while(1) {
        ?
        m_cmd.cmd = VCI_CMD_READ;
        p_vci.send(&m_cmd, m_time.getTime());   // lecture bloquante
        wait(m_rsp_resceived);
        ?
        m_cmd.cmd = VCI_CMD_WRITE;
        p_vci.send(&m_cmd, m_time.getTime());   // écriture non bloquante
        ...
        // lookahead management
        m_counter++ ;
        if (m_counter >= m_lookahead) {
            m_counter = 0 ;
            wait(SC_ZERO_TIME) ;
        } // end if
        m_time.addtime(1) ;
        } // end while
    } // end execLoop()

    //////////////// call-back function
    void  rspReceived(vci_cmd_t *cmd, uint32_t rsp_time)
    {
    if(cmd == VCI_CMD_READ) {
        m_time.set_time(rsp_time + length);
        m_rsp_received.notify(SC_ZERO_TIME) ;
    }
```

```
    } // end rspReceived()
} // end class my_initiator
```

# D) VCI target modeling

In the proposed example, the target handle two types of command : a read burst of 8 words, and a write burst of 8 words. To simplify the model, there is no error management.

The class **my_target** inherit the class **BaseModule**, that is the basis for all TLM-T modules. The class **my_target** contains a member variable **p_vci** of type **VciTargetPort**. This object has a template parameter **<vci_param>** defining the widths of the VCI ADRESS & DATA fields.

## 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 **cmdReceived()** in the example), will be executed each time a VCI command packet is received on the **p_vci** port. The function name is not constrained, but the arguments must respect the following prototype:

```
uint32_t  cmdReceived(vci_cmd_t  *cmd,
                uint32_t  time)
```

For the read and write transactions, the actual data transfer is performed by this **cmdReceived()** function. To avoid multiple data copies, only the pointer on the initiator data buffer is transported in the VCI command pacquet (source buffer for a write transaction, or destination buffer for a read transaction).

## D.2) Sending a VCI response packet

To send a VCI response packet the **cmdReceived()** function must use the **send()** method, that is a member function of the class **VciTargetPort**, and has the following prototype:

```
void  send( vci_rsp_t *cmd,
           uint32_t time)
```

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

## D.3) Target Constructor

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

```
p_vci(?vci?, this, &my_initiator::cmdReceived)
```

## D.4) VCI target example

```
template <typename vci_param>
class my_target : tlmt::BaseModule {
public:
    VciTargetPort<vci_param>  p_vci;
```

```
    ////////////// constructor
    my_target (sc_module_name   name,
              uint32_t  targetIndex,
              uint32_t  latency):
    p_vci(?vci?,this, &my_target::cmdReceived),
    tlmt::BaseModule(name)
    {
    m_latency = latency;
    m_index = targetIndex;
    } // end constructor

private:
    vci_param::data_t  m_data[8];  // local buffer
    uint32_t  m_latency;         // target latency
    uint32_t  m_index;  // target index
    vci_rsp_t  m_rsp;  // paquet VCI réponse

    /////////////// call-back function
    uint32_t  cmdReceived(vci_cmd_t *cmd,
                uint32_t cmd_time)
    {
    if(cmd->cmd == VCI_CMD_WRITE) {
        for(int i = 0 ; i < length ; i++) m_data[i] = cmd->buf[i];
        }
    if(cmd->cmd == VCI_CMD_READ) {
        for(int i = 0 ; i < length ; i++) cmd->buf[i] = m_data[i];
        }
    m_rsp.srcid = cmd->srcid;
    m_rsp.trdid  = cmd->trdid;
    m_rsp.pktid = cmd>pktid;
    m_rsp.length = cmd->length;
    m_rsp.error = 0;
    rsp_time = cmd_time + latency;
    p_vci.send(&m_rsp, rsp_time) ;
    return (rsp_time + cmd->length);
   } // end cmdReceived()
} // end class my_target
```

# E) VCI Interconnect

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 contain a local clock.
- For the response network, there is 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) VCI initiators and targets synchronisations

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 by 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.

As described in Figure 4, when a VCI command packet - sent by the corresponding arbitration thread - is received by a VCI target, two synchronization mechanisms are activated :

- The **cmdReceived()** function sends a VCI response packet with a date to the source initiator, through the **VciVgmn** response network. The corresponding date can be used to update the initiator local clock.
- The **cmdReceived()** function returns a date to the arbitration thread. This date is used to update the arbitration thread local clock.

# F) Interrupt modeling

Interrupts are asynchronous events that are not transported by the VCI network.

As illustrated in Figure 5, each interrupt line is modeled by a specific point to point, uni-directional channel. It use two ports of type **SynchroOutPort** and **SynchroInPort** that must be declared as member variables of the source and destination modules respectively.

## F.1) Source modeling

The source module (named **my_source** in this example) must contain a member variable **p_irq** of type **SynchroOutPort**. To activate, or desactivate an interruption, the source module must use the **send()** method, that is a member function of the **SynchroOutPort** class. Those interrupt packets transport both a Boolean, and a date. The **send()** prototype is defined as follows :

```
void  send( bool  val, uint32_t  time)
```

## F.2) Destination modeling

The destination module (named here **my_processor**) must contain a member variable **p_irq** of type **SynchroInPortt**, and a call-back function (named here **irqReceived()** that is executed when an interrupt packet is received on the **p_irq** port.

A link between the **p_irq** port and the call-back function mus be established by the port constructor in the constructor of the class **my_processor** :

```
p_irq(?irq?, this, &my_processor::irqReceived)
```

In the Parallel Discrete Event Simulation, the pessimistic approach suppose that any PDES process is not allowed to update his local time until he has received messages on all input ports with dates larger than his local time.

Therefore, a SC_THREAD modeling the behavior of a processor containing an **SynchroInPort** should in principle wait a dated packet on this interrupt port before executing instructions. Such behavior would be very inefficient, and could create dead-lock situations.

The recommended policy for handling interrupts is the following:

- The call-back function **irqReceived()** sets the member variables **m_irqpending** and **m_irqtime**, when a interrupt packet is received on the **p_irq** port.
- The **execLoop()** thread must test the **m_irqpending** variable at each cycle (i.e. in each iteration of the infinite loop).
- If there is no interrupt pending, the thread continues execution. If an interrupt is pending, and the interrupt date is larger than the local time, the thread continues execution. If the interrupt date is equal or smaller than the local time, the interrupt is handled.

Such violation of the the pessimistic parallel simulation create a loss of accuracy on the interrupt handling date. This inaccuracy in the TLM-T simulation is acceptable, as interrupts are asynchronous events, and the timing error is bounded by the **m_lookahead** parameter.

# F.3) Processor with interrupt example

```
class my_processor : tlmt::BaseModule {
public:
    SynchroInPort               p_irq;

  ////// constructor
    my_processor (sc_module_name        name,
                  uint32_t  lookahead) :
    p_irq(?irq?, this, &my_initiator::irqReceived),
    m_time(0),
    tlmt::BaseModule(name)
    {
    m_lookahed = lookahead;
    m_counter = 0;
    m_irqset = false;
    SC_THREAD(execLoop);
} // end constructor

private:
    tlmt_time  m_time;  // local clock
    bool  m_irqpendig;  // pending interrupt request
    uint32_t  m_irqtime;  // irq date
    uint32_t  m_counter;  // iteration counter
    uint32_t  m_lookahed;  // lookahead value

    ////////////// thread
    void execLoop()
    {
    while(1) {
        ...
        // test interrupts
        if (m_irqpending && (m_irqtime <= m_time.getTime())) {
```

```
                                        // traitement interrupt
                                        }
                              ...
        // lookahead management
        m_counter++ ;
        if (m_counter >= m_lookahead) {
            m_counter = 0 ;
            wait(SC_ZERO_TIME) ;
            } // end if
        m_time.addtime(1) ;
        } // end while
    } // end execLoop()

///////////////////////////////////////
     void  irqReceived(bool val, sc_time time)
    {
    m_irqpending = val;
    m_irqtime = time;
    } // end irqReceived()
} // end class my_processor
```