Writing efficient TLM-T SystemC simulation models for SoCLib

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

A) Introduction

This document is under development.

This new document describes the modeling rules for writing TLM-T SystemC simulation models for SoCLib. This is the second release, as the modeling rules have been modified to respect the TLM2.0 standard.

In TLM2.0, both the **payload** (defining the actual content of the exchanged packets), and the **phase** (defining the actual communication protocol steps) are template parameters. Therefore, specific **payload** and **phase** types have been defined for SoCLib. Those types are well suited for the VCI/OCP protocol, but they more general and can used to describe any shared-memory architecture using routed network on chip. The TLM-T rules used in SoCLib 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.

Before writing a new model, it is useful to have a 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.

Unlike the initiator, the target module has a purely reactive behaviour and there is no need to use a SC_THREAD : The target behaviour is entirely described by the call-back function, that is executed in the context of the **execLoop() thread 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 contains 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 should use the setter functions **set_time()** and **update_time()** to assign a new time to the PDES 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(soclib::tlmt::vci_cmd_packet<vci_param> *cmd, // pointer to a VCI command packet
          tlmt_core::tlmt_time time);                   // initiator local time
```

The contents of a VCI command packet is 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;   // address on the target side
        uint32_t 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 nwords;                        // 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 optimization.

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), must be declared in the **VciSimpleInitiator** class and 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:

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

The contents of a VCI response packet is defined below:

```
template<typename vci_param>
class vci_rsp_packet
{
public:
        typename vci_param::cmd_t cmd;    // VCI transaction type
        size_t nwords;                    // 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 second parameter, **time**, is of type **tlmt_core::tlmt_time** and corresponds to the time of the response. The third parameter, **private_data** has a default value of **NULL** and is not used when transmitting or receiving VCI packets.

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 reactivates 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**. This allows the target to get information on the initiator that actually sends VCI packets (the initiator local time, the initiator activity, etc).

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

```
/////////////////////////// vci_simple_initiator.h ////////////////////////////////

#ifndef VCI_SIMPLE_INITIATOR_H
#define VCI_SIMPLE_INITIATOR_H

#include <tlmt>
#include "tlmt_base_module.h"
#include "vci_ports.h"

namespace soclib { namespace tlmt {

template<typename vci_param>
class VciSimpleInitiator
    : public soclib::tlmt::BaseModule
{
    tlmt_core::tlmt_thread_context c0;
    sc_core::sc_event m_rsp_received;
    tlmt_core::tlmt_return m_return;
    soclib::tlmt::vci_cmd_packet<vci_param> cmd;

    uint32_t addresses[8];       // address table,
    uint32_t localbuf[8];        // local buffer

    uint32_t m_counter;          // iteration counter
    uint32_t m_lookahead;        // lookahead value
    uint32_t m_index;            // initiator index

protected:
    SC_HAS_PROCESS(VciSimpleInitiator);

public:
    soclib::tlmt::VciInitiator<vci_param> p_vci;

    VciSimpleInitiator( sc_core::sc_module_name name,
                                uint32_t initiator index,
                                uint32_t lookahead );

    tlmt_core::tlmt_return &rspReceived(soclib::tlmt::vci_rsp_packet<vci_param> *pkt,
         const tlmt_core::tlmt_time &time,
         void *private_data);
    void behavior();
};

}}
```

```
#endif

////////////////////////// vci_simple_initiator.cpp ////////////////////////////////

#include "../include/vci_simple_initiator.h"

namespace soclib { namespace tlmt {

#define tmpl(x) template<typename vci_param> x VciSimpleInitiator<vci_param>

tmpl(tlmt_core::tlmt_return&)::rspReceived(soclib::tlmt::vci_rsp_packet<vci_param> *pkt,
        const tlmt_core::tlmt_time &time,
        void *private_data)
{
    if(pkt->cmd == vci_param::VCI_CMD_READ) {
        c0.set_time(time + tlmt_core::tlmt_time(pkt->length));
        m_rsp_received.notify(sc_core::SC_ZERO_TIME) ;
    }
    return m_return;
}

tmpl(void)::behavior()
{

        for(;;) {

                // sending a read VCI packet

                cmd.cmd = vci_param::CMD_READ;  // a VCI read packet
                addresses[0] = 0xBFC00000;      // the start address
                cmd.address = addresses;        // assigned
                cmd.be = 0xF;                   // reading full words
                cmd.contig = true;              // at successive addresses
                cmd.buf = localbuf;             // storing the read results in localbuf
                cmd.length = 8;                 // packet of 8 words
                cmd.srcid = 0;                  // srcid=0
                cmd.trdid = 0;                  // trdid=0
                cmd.pktid = 0;                  // pktid=0

                tlmt_core::tlmt_return ret;        // in case a test on the send function is nee
                ret = p_vci.send(&cmd, c0.time()); // sending the packet
                sc_core::wait(m_rsp_received);     // and waiting for the response packet

                // sending a write VCI packet

                localbuf[0]=0x00112233;         // first, fill the write local buffer with write

                cmd.cmd = vci_param::CMD_WRITE; // then issue the VCI write packet
                addresses[0] = 0x10000000;      // starting with this address
                cmd.address = addresses;
                cmd.be = 0xF;
                cmd.contig = 0;
                cmd.buf = localbuf;
                cmd.length = 1;
                cmd.srcid = 0;
                cmd.trdid = 0;
                cmd.pktid = 0;

                ret = p_vci.send(&cmd, c0.time()); // Don't wait for the answer

                // lookahead management
                m_counter++ ;
                if (m_counter >= m_lookahead) {
                   m_counter = 0 ;
                   sc_core::wait(sc_core::SC_ZERO_TIME) ;
                } // end if
```

```
                // process time= process time+1
                c0.set_time(c0.time()+tlmt_core::tlmt_time(1)) ;

        }
}

tmpl(/**/)::VciSimpleInitiator( sc_core::sc_module_name name,
                                uint32_t initiator index,
                                uint32_t lookahead)
                    : soclib::tlmt::BaseModule(name),
                      m_index(initiator_index),
                      m_lookahead(lookahead),
                      m_counter(0),
                    p_vci("vci", new tlmt_core::tlmt_callback<VciSimpleInitiator,soclib::tlmt::vc
                                        this, &VciSimpleInitiator<vci_param>::rspReceived), &c0
{
        SC_THREAD(behavior);
}

}}
```

D) VCI target modeling

In the proposed example, the target handles 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 **VciSimpleTarget** inherits from the class **BaseModule**. The class **VciSimpleTarget** contains a member variable **p_vci** of type **VciTarget**. This object has a template parameter **<vci_param>** defining the widths of the VCI ADDRESS and 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 **VciSimpleTarget**. 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:

```
tlmt_core::tlmt_return &cmdReceived(soclib::tlmt::vci_cmd_packet<vci_param> *pkt,
         const tlmt_core::tlmt_time &time,
         void *private_data);
```

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 packet (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 **VciTarget**, and has the following prototype:

```
void send(soclib::tlmt::vci_rsp_packet<vci_param> *rsp, // pointer to a VCI response packet
          tlmt_core::tlmt_time time);                   // initiator local time
```

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

D.3) Target Constructor

The constructor of the class **VciSimpleTarget** 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 **VciTarget** constructor must be called with the following arguments:

```
p_vci("vci", new tlmt_core::tlmt_callback<VciSimpleTarget,soclib::tlmt::vci_cmd_packet<vci_param
                                    this, &VciSimpleTarget<vci_param>::cmdReceived))
```

D.4) VCI target example

```
/////////////////////////// vci_simple_target.h ////////////////////////////////

#ifndef VCI_SIMPLE_TARGET_H
#define VCI_SIMPLE_TARGET_H

#include <tlmt>
#include "tlmt_base_module.h"
#include "vci_ports.h"

namespace soclib { namespace tlmt {

template<typename vci_param>
class VciSimpleTarget
    : public soclib::tlmt::BaseModule
{
private:
    tlmt_core::tlmt_return m_return;
    uint32_t m_index;
    uint32_t m_latency;
    soclib::tlmt::vci_rsp_packet<vci_param> rsp;

public:
    soclib::tlmt::VciTarget<vci_param> p_vci;

    VciSimpleTarget(sc_core::sc_module_name name,
              uint32_t  targetIndex,
              uint32_t  latency);

    tlmt_core::tlmt_return &cmdReceived(soclib::tlmt::vci_cmd_packet<vci_param> *pkt,
         const tlmt_core::tlmt_time &time,
         void *private_data);
};

}}

#endif

/////////////////////////// vci_simple_target.cpp ////////////////////////////////

#include "../include/vci_simple_target.h"

namespace soclib { namespace tlmt {

#define tmpl(x) template<typename vci_param> x VciSimpleTarget<vci_param>

tmpl(tlmt_core::tlmt_return&)::cmdReceived(soclib::tlmt::vci_cmd_packet<vci_param> *pkt,
        const tlmt_core::tlmt_time &time,
        void *private_data)
{
    uint32_t m_data[128];
```

```
    if(pkt->cmd == vci_param::VCI_CMD_WRITE) {
        for(int i = 0 ; i < pkt->length ; i++)
                m_data[i] = cmd->buf[i];
        }
    if(pkt->cmd == vci_param::VCI_CMD_READ) {
        for(int i = 0 ; i < pkt->length ; i++)
                cmd->buf[i] = m_data[i];
        }
    rsp.srcid = pkt->srcid;
    rsp.trdid  = pkt->trdid;
    rsp.pktid = pkt->pktid;
    rsp.length = pkt->length;
    rsp.error = 0;
    p_vci.send(&m_rsp, time+tlmt_core::tlmt_time(latency+pkt->length)) ;
    m_return.time=time+tlmt_core::tlmt_time(latency+pkt->length;
    return (m_return);
}

tmpl(/**/)::VciSimpleTarget(sc_core::sc_module_name name,
              uint32_t  targetIndex,
              uint32_t  latency)
        : soclib::tlmt::BaseModule(name),
        m_index(targetIndex),
        m_latency(latency),
        p_vci("vci", new tlmt_core::tlmt_callback<VciSimpleTarget,soclib::tlmt::vci_cmd_packet<v
                                    this, &VciSimpleTarget<vci_param>::cmdReceived))
{
}

}}
```

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 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) VCI initiators and targets synchronizations

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.

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 an associated time to the source initiator, through the **VciVgmn** response network. The corresponding time can be used to update the initiator local clock.
- The **cmdReceived()** function returns a time to the arbitration thread. This time is used to update the arbitration thread local clock.

F) Interrupt modeling

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

As illustrated in Figure 5, each interrupt line is modeled by a specific point to point, uni-directional channel. This channel uses two ports of type **tlmt_core::tlmt_out<bool>** and **tlmt_core::tlmt_in<bool>** that must be declared as member variables of the source and destination modules respectively.

F.1) Source modeling

The source module (named **VciSimpleSourceInterrupt** in this example) must contain a member variable **p_irq** of type **tlmt_core::tlmt_out<bool>**. To activate, or desactivate an interruption, the source module must use the **send()** method, that is a member function of the **tlmt_core::tlmt_out<bool>** class. These interrupt packets transport both a Boolean, and a time. The **send()** prototype is defined as follows :

```
void  send( bool  val, const tlmt_core::tlmt_time &time)
```

F.2) Destination modeling

The destination module (named here **VciProcessor**) must contain a member variable **p_irq** of type **tlmt_core::tlmt_in<bool>**, 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 must be established by the port constructor in the constructor of the class **VciProcessor** :

```
tlmt_core::tlmt_callback < VciProcessor,bool >(this, &VciProcessor < iss_t,
           vci_param >::irqReceived);
```

In the Parallel Discrete Event Simulation, the pessimistic approach relies on the fact that any PDES process is not allowed to update his local time until it has received messages on all its input ports with times greater than its local time.

Therefore, a SC_THREAD modeling the behavior of a processor containing an **tlmt_core::tlmt_in<bool>** should in principle wait a timestamped packet on its interrupt port before executing instructions. However, such a behavior would be very inefficient, and is prone to 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 its execution. If an interrupt is pending, and the interrupt time is greater than the local time, the thread continues execution. If the interrupt time is equal or smaller than the local time, the interrupt is handled.

Such a violation of the the pessimistic parallel simulation principles creates a loss of accuracy on the interrupt handling timestamp. This loss of accuracy 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

```
//////////////////////////// vci_processor.h ////////////////////////////////

namespace soclib { namespace tlmt {

template<typename iss_t,typename vci_param>
class VciProcessor
    : public soclib::tlmt::BaseModule
{
    tlmt_core::tlmt_thread_context c0;
    sc_core::sc_event m_rsp_received;
    tlmt_core::tlmt_return m_return;
    bool  m_irqpendig;  // pending interrupt request
    tlmt_core::tlmt_time  m_irqtime;  // irq date
    uint32_t  m_counter;  // iteration counter
    uint32_t  m_lookahead;  // lookahead value

protected:
    SC_HAS_PROCESS(VciProcessor);

public:
    soclib::tlmt::VciInitiator<vci_param> p_vci;
    tlmt_core::tlmt_in<bool> p_irq;

    VciProcessor( sc_core::sc_module_name name, int id );

    tlmt_core::tlmt_return &rspReceived(soclib::tlmt::vci_rsp_packet<vci_param> *pkt,
                const tlmt_core::tlmt_time &time,
                void *private_data);
    tlmt_core::tlmt_return &irqReceived(bool,
                const tlmt_core::tlmt_time &time,
                void *private_data);
    void execLoop();
};

}}
```

```
/////////////////////////// vci_processor.cpp ///////////////////////////////

#include "../include/vci_processor.h"

namespace soclib
{
  namespace tlmt
  {

#define tmpl(x) template<typename iss_t, typename vci_param> x VciProcessor<vci_param>

    tmpl (tlmt_core::tlmt_return &)::rspReceived (soclib::tlmt:: vci_rsp_packet < vci_param > *p
                                                  const tlmt_core:: tlmt_time & time, void *priv
    {
      if (pkt->cmd == vci_param::CMD_WRITE)
        m_write_error = (pkt->error != 0);
      else
        m_write_error = false;
      if (pkt->cmd == vci_param::CMD_READ)
        m_read_error = (pkt->error != 0);
      else
        m_read_error = false;
      m_rsptime = time + tlmt_core::tlmt_time (pkt->length);
      m_vci_pending = false;
      m_rsp_received.notify (sc_core::SC_ZERO_TIME);

      return m_return;
    }

    tmpl (tlmt_core::tlmt_return &)::irqReceived (bool v, const tlmt_core::
                                                  tlmt_time & time, void *private_data)
    {
      m_irqpending = val;
      m_irqtime = time;

      return m_return;
    }


    tmpl (void)::execLoop ()
        {
                while(1) {
                        ...
                        // test interrupts
                        if (m_irqpending && (m_irqtime <= c0.time()))
                        {
                                // interrupt handling
                        }
                                ...
                        // lookahead management
                        m_counter++ ;
                        if (m_counter >= m_lookahead) {
                                m_counter = 0 ;
                                sc_core::wait(sc_core::SC_ZERO_TIME) ;
                        } // end if
                        c0.set_time(c0.time()+tlmt_core::tlmt_time(1)) ;
                } // end while
        }

  tmpl ( /**/)::VciProcessor(
          sc_core::sc_module_name name,
          int id )
: soclib::tlmt::BaseModule (name),
  m_counter (0),
  m_lookahead (10),
  p_vci("vci",new tlmt_core::tlmt_callback<VciProcessor,vci_rsp_packet<vci_param>*>(
```

```
        this,&VciProcessor<vci_param>::rspReceived),&c0),
  p_irq("irq",new tlmt_core::tlmt_callback<VciProcesspr,bool>(
        this,&VciProcessor<vci_param>::irqReceived))
    {
      SC_THREAD (execLoop);
    }

}}
```