

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\) Lookahead parameter](#)
 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](#)
 2. [F.2\) Arbitration Policy](#)

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. Models complying to these 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 distributed, parallelized simulation engines.

The examples presented below use the VCI/OCF communication protocol selected by the SoCLib project, but the TLM-T approach described here is very flexible, and is not limited to the VCI/OCF 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 **vci_rsp_received()** 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 `vci_cmd_received()`, 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, one to transmit the VCI response packet.

C) VCI Transaction in TLM-T

The TLM2.0 standard allows the user to redefine both the payload and the phases of the transactions. Two classes have been defined in `soclib_vci_types` : a `tlmt_vci_transaction` and a `tlmt_phase`".

The payload of a `tlmt_vci_transaction` contains three groups of information:

- TLM2.0 generic fields
- TLM-T specific fields
- VCI specific fields

```
class tlmt_vci_transaction
{
    ...
private:

    // TLM2.0 related fields and common structure

    sc_dt::uint64      address;
    unsigned char*     data;
    unsigned int       length;
    tlm_response_status response_status;
    unsigned char*     byte_enable;
    unsigned int       streaming_width;

    // TLM-T related fields

    bool*              activity_status_ptr;
    sc_core::sc_time*  local_time_ptr;

    // VCI related fields

    vci_param::cmd_t    cmd;
    vci_param::srcid_t  srcid;
    vci_param::trdid_t  trdid;
    vci_param::pktid_t  pktid;
```

The TLM2.0 compliant accessors allow to set the TLM2.0 related fields, such as the transaction address, the data & byte enable arrays pointers and its associated size in bytes.

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

NB : The char array approach defined by TLM2.0 can degrade the simulation speed, as the existing SoCLib models use `uint32_t` arrays to model both the embedded memories and the caches... Experiments are currently in progress to evaluate the performance degradation incurred by this char <-> uint formatting. It is therefore possible that the types of the data and byte_enable fields of the `tlmt_vci_transaction` will be changed to `uint32*` .

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 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_local_time;           // the initiator local time
...
void addLocalTime(sc_core::sc_time t);    // add an increment 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 boolean member variable **m_activity_status** indicates if the initiator is currently active. It is used by the arbitration threads contained in the **vci_vgmn** interconnect, as described in section F. The corresponding access functions are **setActivity()** and **getActivity()**.

```
bool m_activity_status;
...
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.

The **my_initiator** class contains a member variable **p_vci_init**, of type **tlmt_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
( soclib_vci_types::tlm_payload_type &payload,    // payload
  soclib_vci_types::tlm_phase_type &phase,        // transaction phase (TLMT_CMD)
  sc_core::sc_time &time);                        // local time
```

The first argument is a pointer to the payload, the second represents the phase, 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 **vci_rsp_received()** 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 vci_rsp_received
( soclib_vci_types::tlm_payload_type &payload,      // payload
  soclib_vci_types::tlm_phase_type   &phase,        // transaction phase (TLMT_RSP)
  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 systematically set to `tlm::TLM_COMPLETED`.

In the general case, the actions executed by the interface function depend on both the phase argument, and on the transaction type (defined in the payload). For sake of simplicity, the interface function proposed below does not make any distinction between transaction types.

D.4) 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()** 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::vci_rsp_received);
```

D.5) 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 descheduled. 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.

D.6) VCI initiator example

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

#ifndef __MY_INITIATOR_H__
#define __MY_INITIATOR_H__

#include "tlm.h" // TLM headers
#include "tlmt_transactions.h" // VCI headers
#include "tlmt_simple_initiator_socket.h" // VCI socket
#include "mapping_table.h"

class my_initiator // my_initiator
: public sc_core::sc_module // inherit from SC module base class
{
private:
```

```

//Variables
typedef soclib::tlmt::VciParams<uint32_t,uint32_t,4> vci_param;
sc_core::sc_event m_rspEvent;
sc_core::sc_time m_localTime;
bool m_activity;
uint32_t m_initid;
uint32_t m_counter;
uint32_t m_lookahead;

////////////////////////////////////
// Functions
////////////////////////////////////
void execLoop(void); // initiator thread
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
void setActivity(bool t); // set the activity status (true if the comp
bool getActivity(void); // get the activity state

////////////////////////////////////
// Virtual Fuctions tlm::tlm_bw_transport_if (VCI INITIATOR SOCKET)
////////////////////////////////////

/// Receive rsp from target
tlm::tlm_sync_enum vci_rsp_received // for resp messages
( soclib_vci_types::tlm_payload_type &payload, // payload
  soclib_vci_types::tlm_phase_type &phase, // transaction phase
  sc_core::sc_time &time); // resp time

protected:
    SC_HAS_PROCESS(my_initiator);

public:
    tlmt_simple_initiator_socket<my_initiator,32,soclib_vci_types> p_vci_init; // VCI initiator

    //constructor
    my_initiator( // constructor
                  sc_core::sc_module_name name, // module name
                  const soclib::common::IntTab &index, // VCI initiator index
                  const soclib::common::MappingTable &mt, // mapping table
                  uint32_t lookahead // lookahead
    );

};
#ifdef MY_INITIATOR_H
// my_initiator.cpp
#include "my_initiator.h" // Our header

#ifdef MY_INITIATOR_DEBUG
#define MY_INITIATOR_DEBUG 1
#endif

#define tml(x) x my_initiator

///Constructor
tml (/**/):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
  uint32_t lookahead // lookahead
)
: sc_module(name) // init module name
, p_vci_init("p_vci_init") // vci initiator socket name
{
    //register callback function

```

```

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

// initiator identification
m_initid = mt.indexForId(index);

//lookahead control
m_counter = 0;
m_lookahead = lookahead;

//initialize the local time
m_localTime= 0 * UNIT_TIME;

// initialize the activity variable
setActivity(true);

// register thread process
SC_THREAD(execLoop);
}

////////////////////////////////////
// Functions
////////////////////////////////////
tmpl (sc_core::sc_time)::getLocalTime()
{
    return m_localTime;
}

tmpl (bool)::getActivity()
{
    return m_activity;
}

tmpl (void)::setLocalTime(sc_core::sc_time &t)
{
    m_localTime=t;
}

tmpl (void)::addLocalTime(sc_core::sc_time t)
{
    m_localTime= m_localTime + t;
}

tmpl (void)::setActivity(bool t)
{
    m_activity =t;
}

tmpl (void)::execLoop(void)    // initiator thread
{
    soclib_vci_types::tlm_payload_type payload;
    soclib_vci_types::tlm_phase_type phase;
    sc_core::sc_time sendTime;
    unsigned char data[32];
    unsigned char byte_enable[32];
    int pktid = 0;
    int nbytes = 4; // 1 word of 32 bits

    uint32_t int_data = 12345678;
    std::ostringstream name;
    name << " " << int_data;
    std::cout << "NAME = " << std::dec << name << std::endl;

    for(int i=0; i<nbytes; i++)
        byte_enable[i] = TLMT_BYTE_ENABLED;

    for(int i=0; i<32; i++)

```

```

    data[i] = 0x0;

data[0]='a';
data[1]='b';
data[2]='c';
data[3]='d';
data[4]='\0';

std ::cout<< "DATA = " << data << std::endl;

while ( 1 ){
    addLocalTime(10 * UNIT_TIME);

    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_initid);
    payload.set_trd_id(0);
    payload.set_pkt_id(pktid);

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

#ifdef MY_INITIATOR_DEBUG
    std::cout << "[INITIATOR " << m_initid << "] send cmd packet id = " << payload.get_pkt_id()
#endif

    p_vci_init->nb_transport_fw(payload, phase, sendTime);
    wait(m_rspEvent);

#ifdef MY_INITIATOR_DEBUG
    std::cout << "[INITIATOR " << m_initid << "] receive rsp packet id = " << payload.get_pkt_id()
#endif

    pktid++;

    addLocalTime(10 * UNIT_TIME);

    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_read();
    payload.set_src_id(m_initid);
    payload.set_trd_id(0);
    payload.set_pkt_id(pktid);

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

#ifdef MY_INITIATOR_DEBUG
    std::cout << "[INITIATOR " << m_initid << "] send cmd packet id = " << payload.get_pkt_id()
#endif

    p_vci_init->nb_transport_fw(payload, phase, sendTime);
    wait(m_rspEvent);

#ifdef MY_INITIATOR_DEBUG
    std::cout << "[INITIATOR " << m_initid << "] receive rsp packet id = " << payload.get_pkt_id()
#endif
#endif

```

```

        pktid++;

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

    } // end while true
    setActivity(false);
} // end initiator_thread

////////////////////////////////////
// Virtual Functions  tlm::tlm_bw_transport_if (VCI SOCKET)
////////////////////////////////////
/// receive the response packet from target socket

tmpl (tlm::tlm_sync_enum)::vci_rsp_received          //
( soclib_vci_types::tlm_payload_type &payload,      // VCI payload
  soclib_vci_types::tlm_phase_type    &phase,       // tlm phase
  sc_core::sc_time                    &rspTime      // time
)
{
    switch(phase){
    case soclib::tlmt::TLMT_RSP :
        setLocalTime(rspTime);
        m_rspEvent.notify(0 * UNIT_TIME);
        break;
    case soclib::tlmt::TLMT_INFO :
        payload.set_local_time_ptr(&m_localTime);
        payload.set_activity_ptr(&m_activity);
        break;
    }
    return tlm::TLM_COMPLETED;
} // end backward nb transport

```

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 **tlmt_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 **vci_cmd_received()** 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 vci_cmd_received
( soclib_vci_types::tlm_payload_type &payload,          // payload
  soclib_vci_types::tlm_phase_type   &phase,           // transaction phase (TLMT_CMD)
  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 systematically set to `tlm::TLM_COMPLETED`.

E.3) Sending a VCI response packet

To send a VCI response packet the call-back function `vci_cmd_received()` uses the `nb_transport_bw()` method, defined by TLM2.0, 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 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 vci_cmd_received (
    soclib_vci_types::tlm_payload_type &payload,
    soclib_vci_types::tlm_phase_type   &phase,
    sc_core::sc_time                   &time)
{
    ...
    payload.set_response_status(soclib::tlmt::TLM_OK_RESPONSE);
    phase = soclib::tlmt::TLMT_RSP;
    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 `vci_cmd_received()` 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);

```

E.5) VCI target example

```

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

```

```

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

```

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 initiators 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 for each output port is round-robin.



F.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 time. This time represents the target local time.
- 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 :



F.2) Arbitration Policy

As described above, there is one **cmd_arbitration** thread associated to each VCI target. This thread is in charge of selecting one timed request between all possible requesters, and to forward it to the target. According to the PDES principles, the arbitration thread must select the request with the smallest timestamp. 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 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. Any arbitration thread receiving a timed request is resumed. It must obtain an up to date timing & activity information for all its input channels before making any decision. To do that, the **m_local_time** and **m_activity_status** variables of all VCI initiators are considered as public variables, that can be accessed (read only) by all arbitration threads. The arbitration policy is the following : The arbitration thread scans all its input channels, and selects the smallest time between the active initiators. If there is a request, this request is forwarded to the target, and the arbitration thread local time is updated. If there is no request from this initiator, the thread is descheduled and will be resumed when it receives a new request.

For efficiency reasons, in this implementation, each arbitration thread constructs - during elaboration of the simulation - two local array of pointers (indexed by the input channel index) to access the **m_local_time** and **m_activity_status** variables of all VCI initiators. To get this information, each arbitration thread uses the

nb_transport_bw() function on all its VCI target ports, with a dedicated value for the phase called **soclib::tlmt::TLMT_INFO**. The payload argument refers to the same **tlmt_vci_transaction** object as the two other phase values (**soclib::tlmt::TLMT_CMD** and **soclib::tlmt::TLMT_RSP**).

```
for (size_t i=0;i<m_nbinit;i++) {  
    phase    = soclib::tlmt::TLMT_INFO;  
    m_RspArbCmdRout[i]->p_vci->nb_transport_bw(payload, phase, rspTime);  
    m_array[i].ActivityStatus = payload.get_activity_ptr();  
    m_array[i].LocalTime = payload.get_local_time_ptr();  
}
```

As the net-list of the simulated platform must be explicitly defined before constructing the LocalTime and ActivityStatus arrays, the vgm hardware component provides an utility function **fill_time_activity_arrays()** that must be called in the SystemC top-cell, before starting the simulation.