

Writing efficient TLM-T SystemC simulation models for SoCLib

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

1. A) Introduction
2. B) VCI Communication between a single initiator and a single target
3. C) 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.5) TLM-T initiator example
4. D) 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) TLM-T target example

A) Introduction

This document describes the modeling rules for writing TLM-T SystemC simulation models for SoCLib. Those rules enforce the PDES (Parallel Discrete Event Simulation) principles. Each PDES process involved in the simulation has its own, local time, and processes synchronize through timed messages. Models complying with those rules can be used with the "standard" OSCI simulation engine (SystemC 2.x), but can be used also with others simulation engines, especially distributed, parallelized simulation engines.

Besides you may also want to follow the general SoCLib rules.

B) VCI Communication between a single initiator and a single target

Figure 1 presents a minimal system containing one single initiator, and a single target. In the proposed example, the initiator module doesn't contain any parallelism, and can be modeled by a single SC_THREAD, describing a single PDES process. The activity of the **my_initiator** module is described by the SC_THREAD **execLoop()**, that contains an infinite loop. The variable **m_time** represents the PDES process local time.

Contrary to the initiator, the target module has a purely reactive behaviour. There is no need to use a SC_THREAD to describe the target behaviour : A simple method is enough.

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) Initiator Modeling

In the proposed example, the initiator module is modeled by the **my_initiator** class. This class inherits the **Tlmt::BaseModule** class, that is the basis for all TLM-T modules. As there is only one thread in **my_initiator**, there is only one member variable **time** of type **tlmt_time**. *This object can be accessed through the **getTime()**, **addTime()** and **setTime()** methods. The **execLoop()** method, describing the initiator activity must be declared as a*

member function of the my_initiator class.

C.1) Sending a VCI command packet

The class **my_initiator** must contain a member variable **p_vci**, of type **VciInitiatorPort**. This object has a template parameter **<vci_param>** defining the widths of the VCI ADDRESS & DATA fields.

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

```
void cmdSend(    vci_cmd_t      *cmd,    // VCI command packet
                sc_time        time);    // initiator local time
```

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

```
class vci_cmd_t {
vci_command_t    cmd;                // VCI transaction type
vci_address_t    *address;           // pointer to an array of addresses on the target side
uint32_t         *be;                // pointer to an array of byte_enable si
bool             contig;              // contiguous addresses (when true)
vci_data_t       *buf;               // pointer to the local buffer on the initiator
uint32_t         length;              // number of words in the packet
bool             eop;                // end of packet marker
uint32_t         srcid;              // SRCID VCI
uint32_t         trdid;              // TRDID VCI
uint32_t         pktid;              // PKTID VCI
}
```

C.2) Receiving a VCI response packet

C.3) Initiator Constructor

C.4) Lookahead parameter

C.5) TLM-T 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) ,
        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

////////// thread
void execLoop()
{
    while(1) {
        ?
        m_cmd.cmd = VCI_CMD_READ;
p_vci.cmdSend(&m_cmd, m_time.get_time()); // lecture bloquante
        p_vci.wait();
        ?
        m_cmd.cmd = VCI_CMD_WRITE;
        p_vci.send(VCI_CMD_WRITE,?);
p_vci.cmdSend(&m_cmd, m_time.get_time()); // 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, sc_time rsp_time)
{
    if(cmd == VCI_CMD_READ) {
m_time.set_time(rsp_time + length);
p_vci.notify() ;
    }

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

```

D) Target Modeling

D.1) Receiving a VCI command packet

D.2) Sending a VCI response packet

D.3) Target Constructor

D4) TLM-T target example

```

Cible TLM-T

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,
sc_time          latency):

```

```

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

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

////////// call-back function
sc_time cmdReceived( vci_cmd_t *cmd,
sc_time 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.rspSend(&m_rsp, rsp_time) ;
        return (rsp_time + (sc_time)cmd->length);
        } // end cmdReceived()

} // end class my_target

```