Writing efficient TLM-T SystemC simulation models for SoCLib

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

A) Introduction

This manual 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 is 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 contains 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 contain 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-directionnal channel, encapsulating two separated uni-directionnal 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 inherit 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 ADRESS & 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
```

As VCI signals can have variable widths, all VCI components must be defined with templates. The caba/interface/vci_param.h file contains the definition of the VCI parameters object. This object must be passed as a template parameter to the component.

A typical VCI component declaration is:

```
#include "caba/util/base_module.h"
#include "caba/interface/vci_target.h"

namespace soclib { namespace caba {

template<typename vci_params>
class VciExampleModule
    : soclib::caba::BaseModule
{

};
}}
```

The SystemC top cell defining the system architecture must include the following file, defining the advanced VCI signals :

- caba/interface/vci_signals.h.

A SoCLib hardware component that has no VCI interface should use a dedicated VCI wrapper in order to be connected to the VCI interconnect.

B3) Address space segmentation

In a shared memory architecture, the address space segmentation (or memory map) is a global characteristic of the system. This memory map must be defined by the system designer, and is used by both software, and hardware components.

Most hardware components use this memory map:

- VCI interconnect components contain a *routing table* used to decode the VCI address MSB bits and route VCI commands to the proper targets.
- VCI target components must be able to check for segmentation violation when receiving a command packet. Therefore, the base address and size of the segment allocated to a given VCI target must be *known* by this target.
- A cache controller supporting uncached segments can contain a *cacheability table* addressed by the VCI address MSB bits.

In order to simplify the memory map definition, and the hardware component configuration, a generic object, called *mapping table* has been defined in common/mapping_table.h. This is an associative table of memory segments. Any segment must be allocated to one single VCI target. The segment object is defined in common/segment.h, and contains five attributes:

```
const std::string   m_name;           // segment's name
addr_t              m_base_address;   // base address
size_t              m_size;           // size (bytes)
IntTab              m_target_index;   // VCI target index
bool                m_cacheability;   // cacheable
```

Any hardware component using the memory map should have a constant reference to the mapping table as constructor argument.

B4) Component definition

The component *XXX.h* file contains the following informations

Interface definition A typical VCI target component will contain the following ports:

```
sc_in<bool>                        p_resetn;
sc_in<bool>                        p_clk;
soclib::caba::VciTarget<vci_param>  p_vci;
```

Internal registers definition

All internal registers should be defined with the *sc_signal* type.

This point is a bit tricky: It allows the model designer to benefit from the delayed update mechanism associated by SystemC to the sc_signal type. When a single module contains several interacting FSMs, it helps to write the

`Transition()`, as the registers values are not updated until the exit from the transition function. Conversely, any member variable declared with the `sc_signal` type is considered as a register.

A typical VCI target will contain the following registers :

```
sc_signal<int>                            r_vci_fsm;
sc_signal<typename vci_param::trdid_t>  r_buf_trdid;
sc_signal<typename vci_param::pktid_t>  r_buf_srcid;
sc_signal<typename vci_param::srcid_t>  r_buf_srcid;
```

`typename vci_param::trdid_t` and others are generically-defined VCI field types

Structural parameters definition

All structural parameters should be be defined as member variables. The values are generally defined by a constructor argument. Instance name is stored in soclib::common::BaseModule, inherited by soclib::caba::BaseModule. For example, a VCI target will contain a reference to the the assigned segment, in order to check possible segmentation errors during execution.

```
const soclib::common::Segment   m_segment;
```

B5) Constructor & destructor

Any hardware component must have an instance name, and most SoCLib component must have a VCI index. Moreover, generic simulation models can have structural parameters. The parameter values must be defined as constructor arguments, and used by the constructor to configure the hardware ressources. A constructor argument frequently used is a reference on the soclib::common::MappingTable, that defines the segmentation of the system address space. A typical VCI component will have the following constructor arguments:

```
VciExampleModule(
    sc_module_name                      insname,
    const soclib::common::IntTab       &index,
    const soclib::common::MappingTable &mt);
```

In this example, the first argument is the instance name, the second argument is the VCI target index, and the third argument is the mapping table.

Moreover, the constructor must define the sensitivity list of the Transition(), genMoore() and genMealy() methods, that are described below.

- sensitivity list of the transition() method contains only the clock rising edge.
- sensitivity list of the genMoore() method contains only the clock falling edge.
- sensitivity list of the genMealy() method contains the clock falling edge, as well as all input ports thare in the support of the Mealy generation function.

Be careful: the constructor should NOT initialize the registers. The register initialization must be an hardware mechanism explicitly described in the Transition function on reset condition.

B6) member functions

The component behaviour is described by simple member functions. The type ot those methods (Transition, genMoore, or genMealy) is defined by the sensitivity lists, as specified in B5.

transition() method

For each hardware component, there is only one `Transition()` method. It is called once per cycle, as the sensitivity list contains only the clock rising edge. This method computes the next values of the registers (variables that have the `sc_signal` type). No output port can be assigned in this method. Each register should be assigned only once.

genMoore() method

For each hardware component, there is only one `genMoore()` method. It is called once per cycle, as the sensitivity list contains only the clock falling edge. This method computes the values of the Moore output ports. (variables that have the `sc_out` type). No register can be assigned in this method. Each output port can be assigned only once. No input port can be read in this method

genMealy() methods

For each hardware component, there is zero, one or several `genMealy()` methods (it can be useful to have one separated `gemealy()` method for each output port). These methods can be called several times per cycle. The sensitivity list can contain several input ports. This method computes the Mealy values of the ouput ports, using only the register values and the input ports values. No register can be assigned in this method. Each output port can be assigned only once.

C) Complete example

C1) Component definition

Let's take the soclib::caba::VciLocks component definition and comment it.

```
#include <systemc.h>
#include "caba/util/base_module.h"
#include "caba/interface/vci_target.h"
#include "common/mapping_table.h"

// Have this component in the soclib::caba namespace
namespace soclib { namespace caba {

// Here we pass the VCI parameters as a template argument. This is intended because VCI paramete
// change data type widths, therefore change some compile-time intrinsics
template<typename vci_param>
class VciLocks
    : public soclib::caba::BaseModule
{

    // We have only one FSM in this component. It handles the
    // VCI target port. The states are:
    enum vci_target_fsm_state_e {
        IDLE,
        WRITE_RSP,
        READ_RSP,
        ERROR_RSP,
    };

    // The register holding the FSM state:
    sc_signal<int> r_vci_fsm;

    // Some registers used to save useful data between command & response
    sc_signal<typename vci_param::srcid_t> r_buf_srcid;
    sc_signal<typename vci_param::trdid_t> r_buf_trdid;
    sc_signal<typename vci_param::pktid_t> r_buf_pktid;
    sc_signal<typename vci_param::eop_t>   r_buf_eop;
```

```
    sc_signal<bool>                        r_buf_value;

    // Pointer on the table of locks (allocated in the constructor)
    sc_signal<bool>             *r_contents;

    // The segment assigned to this peripheral
    soclib::common::Segment m_segment;

protected:
    // Mandatory SystemC construct
    SC_HAS_PROCESS(VciLocks);

public:
    // The ports
    sc_in<bool> p_resetn;
    sc_in<bool> p_clk;
    soclib::caba::VciTarget<vci_param> p_vci;

    // Constructor & descructor, explained above
    VciLocks(
        sc_module_name insname,
        const soclib::common::IntTab &index,
        const soclib::common::MappingTable &mt);
    ~VciLocks();

private:
    // The FSM functions
    void transition();
    void genMoore();
};

// Namespace closing
}}
```

C2) Component implementation

Here is the soclib::caba::VciLocks component implementation:

```
#include "caba/target/vci_locks.h"

// Namespace, again
namespace soclib { namespace caba {

// This macro is an helper function to factor out the template parameters
// This is useful in two ways:
// * It makes the syntax clearer
// * It makes template parameters changes easier (only one line to change them all)
// x is the method's return value
#define tmpl(x) template<typename vci_param> x VciLocks<vci_param>

// The /**/ is a hack to pass no argument to a macro taking one. (constructor has no
// return value in C++)
tmpl(/**/)::VciLocks(
    sc_module_name insname,
    const soclib::common::IntTab &index,
    const soclib::common::MappingTable &mt)
// This is the C++ construct for parent construction and
// member variables initialization.
// We initialize the BaseModule with the component's name
    : soclib::caba::BaseModule(insname),
// and get the segment from the mapping table and our index
      m_segment(mt.getSegment(index))
{
```

```
    // There is one lock every 32-bit word in memory. We
    // allocate an array of bool for the locks
    r_contents = new sc_signal<bool>[m_segment.size()/4];

    // Sensitivity list for transition() and genMoore(), no genMealy()
    // in this component
    SC_METHOD(transition);
    dont_initialize();
    sensitive << p_clk.pos();

    SC_METHOD(genMoore);
    dont_initialize();
    sensitive << p_clk.neg();
}

tmpl(/**/)::~VciLocks()
{
    // Here we must delete dynamically-allocated data...
    delete [] r_contents;
}

tmpl(void)::transition()
{
    // On reset condition, we initialize the component,
    // from FSMs to internal data.
    if (!p_resetn) {
        for (size_t i=0; i<m_segment.size()/4; ++i)
            r_contents[i] = false;
        r_vci_fsm = IDLE;
        return;
    }

    // We are not on reset case.

    // Take the address, transform it into an index in our locks table.
    typename vci_param::addr_t address = p_vci.address.read();
    uint32_t cell = (address-m_segment.baseAddress())/4;

    // Implement the VCI target FSM
    switch (r_vci_fsm) {
    case IDLE:
        if ( ! p_vci.cmdval.read() )
            break;
        /*
         * We only accept 1-word request packets
         * and we check for segmentation violations
         */
        if ( ! p_vci.eop.read() ||
             ! m_segment.contains(address) )
            r_vci_fsm = ERROR_RSP;
        else {
            switch (p_vci.cmd.read()) {
            case VCI_CMD_READ:
                r_buf_value = r_contents[cell];
                r_contents[cell] = true;
                r_vci_fsm = READ_RSP;
                break;
            case VCI_CMD_WRITE:
                r_contents[cell] = false;
                r_vci_fsm = WRITE_RSP;
                break;
            default:
                r_vci_fsm = ERROR_RSP;
                break;
            }
        }
```

```
        r_buf_srcid = p_vci.srcid.read();
        r_buf_trdid = p_vci.trdid.read();
        r_buf_pktid = p_vci.pktid.read();
        r_buf_eop = p_vci.eop.read();
        break;

    // In those states, we only wait for response to be accepted.
    case WRITE_RSP:
    case READ_RSP:
    case ERROR_RSP:
        if ( p_vci.rspack.read() )
            r_vci_fsm = IDLE;
        break;
    }
}

tmpl(void)::genMoore()
{
    // This is an helper function defined in the VciTarget port definition
    p_vci.rspSetIds( r_buf_srcid.read(), r_buf_trdid.read(), r_buf_pktid.read() );

    // Depending on the state, we give back the expected response.
    switch (r_vci_fsm) {
    case IDLE:
        p_vci.rspNop();
        break;
    case WRITE_RSP:
        p_vci.rspWrite( r_buf_eop.read() );
        break;
    case READ_RSP:
        p_vci.rspRead( r_buf_eop.read(), r_buf_value.read() );
        break;
    case ERROR_RSP:
        p_vci.rspError( r_buf_eop.read() );
        break;
    }

    // We only accept commands in Idle state
    p_vci.cmdack = (r_vci_fsm == IDLE);
}
}}
```

Component instanciation could be (`template_inst.cc`):

```
#include "caba/target/vci_locks.cc"
template class soclib::caba::VciLocks<soclib::caba::VciParams<4,1,32,1,1,1,8,1,1,1> >;
```

Command line:

```
g++ -c -o obj.o -I/path/to/soclib/systemc/src -I/path/to/soclib/systemc/include template_inst.cc
```