Getting Started with TLM-2.0

A Series of Tutorials based on a set of Simple, Complete Examples

John Aynsley, Doulos, June 2008

Tutorial 1 – Sockets, Generic Payload, Blocking Transport

Introduction

The TLM-2.0 transaction level modeling standard from the Open SystemC Initiative (OSCI) was released on 9th June 2008. The official release kit includes comprehensive documentation and a growing set of examples, and can be obtained from the OSCI website, www.systemc.org.

The documentation in the official release kit is good; we know, because we wrote much of it! This tutorial is complementary to the documentation in the release, and will help get you started by providing a tutorial-style introduction to the basic features of TLM-2.0 and how they fit together. This tutorial assumes that you know SystemC and that you know the basics of transaction-level modeling. A knowledge of the OSCI TLM 1.0 standard would be a great starting point, but is not essential.

This tutorial is accompanied by a set of source files that you can download. The examples can be run using SystemC 2.2.0 and TLM-2.0, both available from www.systemc.org. The examples can also be run using SystemC 2.1v1, but you would not want to use an out-of-date version, would you? The examples cannot be run with TLM2.0-draft-1 or TLM2.0-draft-2, which are incompatible with the final release. The only other thing you need is a supported C++ compiler. Alternatively, you can use a dedicated SystemC simulator, although you will have to pay real money for one of those.

Modeling Concepts

Transaction level modeling in SystemC involves communication between SystemC processes using function calls. The focus of TLM is on the communication between the processes rather than the algorithms performed by the processes themselves, so the processes shown in this tutorial will be rather trivial. We assume that in a model of system behavior, some of the SystemC processes will produce data, others will consume data, some will initiate communication, others will passively respond to communication initiated by others.

The focus of OSCI TLM-2.0 in particular is the modeling of on-chip memory-mapped busses. This does not mean that TLM-2.0 is dedicated exclusively to memory-mapped busses, just that this is where most of the features are focussed. TLM-2.0 has a layered structure, with the lower layers being more flexible and general, and the upper layers being specific to bus modeling. In future, the standard may be re-oriented toward other styles of communication as they emerge, the obvious direction being network-on-chip (NoC) architectures.

As a starting point, we will assume you are familiar with the concepts of the module, port, process, channel, interface and event in SystemC. If not, you could start with the SystemC tutorial on www.doulos.com/knowhow/systemc. TLM-2.0 involves having processes embedded within separate modules communicating using interface method calls through ports and exports.

Initiators, Targets, and Sockets

In TLM-2.0, an initiator is a module that initiates new transactions, and a target is a module that responds to transactions initiated by other modules. A transaction is a data structure (a C++ object) passed between initiators and targets using function calls. The same module can act both as an initiator and as a target, and this would typically be the case for a model of an arbiter, a router, or a bus.

In order to pass transactions between initiators and targets, TLM-2.0 uses sockets. An initiator sends transactions out through an initiator socket, and a target receives incoming transactions through a target socket. A module that merely forwards transactions without modifying their content is known as an interconnect component. An interconnect component would have both a target socket and an initiator socket.

Now let us look at some SystemC code. A TLM-2.0 model needs to include the standard SystemC header, the header “tlm.h”, and any other specific headers that you use from the kit. In this case we are using two sockets from the utilities directory (tlm_utils). You need to point the include path of your C++ compiler (or makefile) to the ./include directory in the release.

```
#define SC_INCLUDE_DYNAMIC_PROCESSES

#include "systemc"
using namespace sc_core;
using namespace sc_dt;
using namespace std;

#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"
```

When using the OSCI simulator, it is necessary to define the macro SC_INCLUDE_DYNAMIC_PROCESSES when using certain parts of the TLM-2.0 kit, in particular the simple sockets, the reason being that these particular sockets spawn dynamic processes.

Now we declare initiator and target modules, where the initiator module will generate transactions, and the target module will represent a simple memory. The transactions generated by the initiator will read from or write to the memory.

```
struct Initiator: sc_module
{...};

struct Memory: sc_module
{...};
```

We also have to connect up the module hierarchy:

```
SC_MODULE(Top)
{
  Initiator *initiator;
  Memory    *memory;

  SC_CTOR(Top)
  {
    initiator = new Initiator("initiator");
    memory    = new Memory   ("memory");

    initiator->socket.bind( memory->socket );
  }
};
```

The top-level module of the hierarchy instantiates one initiator and one memory, and binds the initiator socket on the initiator to the target socket on the target memory. The sockets encapsulate everything you need for two-way communication between modules, including ports and exports for both directions of communication. One initiator socket is always bound to one target socket. An initiator socket is actually an **sc_port** that has an **sc_export** on the side, whereas a target socket is actually an **sc_export** that has an **sc_port** on the side. The bind operator of the socket class binds port-to-export in both directions with a single function call. This convenience is a feature of sockets. (It is also possible to bind sockets hierarchically up and down a nested module hierarchy, but we will not worry about that right now.)

For completeness, when using the OSCI simulator, you will also need the following **sc_main** function:

```
int sc_main(int argc, char* argv[])
{
  Top top("top");
  sc_start();
  return 0;
}
```

Within the initiator and memory modules, the initiator and target sockets have to be declared and constructed explicitly, as follows:

```
struct Initiator: sc_module
{
  tlm_utils::simple_initiator_socket<Initiator> socket;

  SC_CTOR(Initiator) : socket("socket")
  {
    ...
};

struct Memory: sc_module
{
  tlm_utils::simple_target_socket<Memory> socket;

  SC_CTOR(Memory) : socket("socket")
  {
    ...
};
```

All TLM-2.0 declarations are in one of the two C++ namespace **tlm** or **tlm_utils**. Just to be clear, we will qualify all such names explicitly throughout the examples. Note the names of the socket types, **simple_initiator_socket<Initiator>** and **simple_target_socket<Memory>**. The first socket template argument specifies the typename of the parent module. There are other socket template arguments, but they are not shown here since we are allowing them to take their default values. The other arguments effectively allow us to specific the width of the socket and the type of the transactions passed through the socket.

The simple initiator and target sockets are so-called because they are simple to use. They are utility classes derived from two underlying socket types **tlm_initiator_socket** and **tlm_target_socket**. You do not need to know about these latter two base classes unless you want to create your own convenience sockets, which is a useful but more advanced coding technique. For now we will keep with the simple sockets, which makes the initiator and target relatively easy to code. Strictly speaking, it is the underlying base classes that are key to interoperability, not the utility sockets. The classes in the **tlm_utils** namespace exist for convenience and productivity; only the classes in the **tlm** namespace are actually essential for interoperability.

In this example, the initiator will communicate with the target memory using the blocking transport interface, so the target needs to implement a single method named **b_transport**. When using the simple target socket, this is done by having the target register a callback method with the socket as follows:

```
socket.register_b_transport(this, &Memory::b_transport);
```

All the target now has to do is to provide an implementation for the **b_transport** method, which we will describe below.

The Generic Payload and Blocking Transport

The default transaction type for the socket classes, implied in the absence of any template arguments, is **tlm_generic_payload**. The generic payload is an important part of the TLM-2.0 standard because it is another of the keys to achieving interoperability between transaction level models. The generic payload serves two closely-related purposes. It can be used as a general-purpose transaction type for abstract memory-mapped bus modeling when you are not concerned with the exact details of any particular bus protocol, offering immediate interoperability between models off-the-shelf. Alternatively, the generic payload can be used as the basis for modeling a wide range of specific protocols at a more detailed level, the beauty of this approach being that it is relatively easy to bridge between different protocols when both are built on top of the same generic payload type.

Our initiator module has a thread process to generate a stream of generic payload transactions.

```
SC_CTOR(Initiator) : socket("socket")
{
  SC_THREAD(thread_process);
}

void thread_process()
{
  tlm::tlm_generic_payload* trans = new tlm::tlm_generic_payload;
  sc_time delay = sc_time(10, SC_NS);

  for (int i = 32; i < 96; i += 4)
  {
    ...
    socket->b_transport( *trans, delay );
    ...
  }
}
```

The transaction is sent through the socket using the **b_transport** method of the TLM-2.0 blocking transport interface, which passes its transaction argument by reference and has no return value. The initiator is responsible for allocating and deleting storage for the transaction. The **b_transport** call also carries a timing annotation, which should be added to the current simulation time (as returned by **sc_time_stamp**) to determine the time at which the transaction is to be processed. The timing annotation is active on both the call to and the return from the **b_transport** method.

```
tlm::tlm_command cmd = static_cast<tlm::tlm_command>(rand() % 2);
if (cmd == tlm::TLM_WRITE_COMMAND) data = 0xFF000000 | i;

trans->set_command( cmd );
trans->set_address( i );
trans->set_data_ptr( reinterpret_cast<unsigned char*>(&data) );
trans->set_data_length( 4 );
trans->set_streaming_width( 4 );
trans->set_byte_enable_ptr( 0 );
trans->set_dmi_allowed( false );
trans->set_response_status( tlm::TLM_INCOMPLETE_RESPONSE );

socket->b_transport( *trans, delay );
```

Each generic payload transaction has a standard set of bus attributes: command, address, data, byte enables, streaming width, and response status. The generic payload also carries a DMI hint and extensions. Although each attribute has a default value, it is recommended practice to set at least 8 of the 10 attributes explicitly before passing the transaction to an interface method call, the reason being that transaction objects are typically reused from a pool.

The generic payload supports two commands, read and write. Here, the command attribute is set to read or write at random.

The address attribute is the lowest address value to which data is to be read or written. Here, the address is set to the loop index.

The data pointer attribute points to a data buffer within the initiator, and the data length attribute gives the length of the data array in bytes. Here, data length is set to 4 bytes. In the case of a write command, the data will be copied from the data array to the target, and in the case of a read command, copied from the target to the data array. In either case, the actual copy is executed at the target.

The streaming width attribute specifies the width of a streaming burst where the address repeats itself. For a non-streaming transaction, the streaming width should equal the data length, as is the case here. Although the default value of the streaming width attribute is 0, a value of 0 is not permitted when a transaction comes to be sent through an interface method call. This same principle applies to the data pointer and data length attributes.

The byte enable pointer is set to 0 to indicate that byte enables are unused. There is also a byte enable length attribute, which is not set here because with the pointer set to 0 it would be ignored.

The **set_dmi_allowed** method sets the DMI hint, which should always be initialized to false. The DMI hint attribute may be set by the target to indicate that it supports the direct memory interface.

The response status should always be initialized to a value of TLM_INCOMPLETE_RESPONSE. The response status may be set by the target.

The tenth generic payload attribute not mentioned above is an array of extensions. Extensions will be discussed in a later tutorial. By default, any extensions may be ignored by an initiator or target.

The blocking transport method is implemented in the target memory. First, the set of six attributes that cannot be ignored are extracted from the generic payload transaction. (The remaining attributes are the DMI hint and response status which are set by the target, the byte enable length which can be ignored if byte enables are unused, and the extensions which can anyway be ignored.)

```
virtual void b_transport( tlm::tlm_generic_payload& trans, sc_time& delay )
{
  tlm::tlm_command cmd = trans.get_command();
  sc_dt::uint64    adr = trans.get_address() / 4;
  unsigned char*   ptr = trans.get_data_ptr();
  unsigned int     len = trans.get_data_length();
  unsigned char*   byt = trans.get_byte_enable_ptr();
  unsigned int     wid = trans.get_streaming_width();
```

Next, the attributes are checked to ensure that the initiator is not trying to use features that the target cannot support. In this case, the target memory does not support byte enables, streaming width, or burst transfers. The following statement also checks that the address is not out-of-range. If the transaction cannot be executed, the SystemC report handler is called to generate an error.

```
  if (adr >= sc_dt::uint64(SIZE) || byt != 0 || len > 4 || wid < len)
    SC_REPORT_ERROR("TLM-2",
                    "Target does not support given generic payload transaction");
```

Of course, if the target is unable to support certain features of the generic payload this will limit interoperability, but at least the set of features is well-defined and there are standard obligations on the target to check and report incompatibility.

The target then implements the read and write command by copying data to or from the data area in the initiator. Regarding endianness, the rule is that the generic payload takes the same endianness as the host computer. As long as the target memory is also modeled using host endianness, the data copy can be done using **memcpy**:

```
  if ( cmd == tlm::TLM_READ_COMMAND )
    memcpy(ptr, &mem[adr], len);
  else if ( cmd == tlm::TLM_WRITE_COMMAND )
    memcpy(&mem[adr], ptr, len);
```

The final act of the blocking transport method is to set the response status attribute of the generic payload to indicate the successful completion of the transaction. If not set, the default response status would indicate to the initiator that the transaction is incomplete.

```
  trans.set_response_status( tlm::TLM_OK_RESPONSE );
```

Timing Annotation

Since the blocking transport method is only modeling the functionality of the target and not modeling any timing detail, it simply ignores the value of the delay argument and returns it to the initiator untouched.

After calling **b_transport**, the initiator checks the response status:

```
if (trans->is_response_error() )
  SC_REPORT_ERROR("TLM-2", "Response error from b_transport");
```

The initiator now needs to realize any annotated timing. Since this model is only concerned with functionality, it may continue to accumulate delays ad infinitum. The idea is for the simulation model to compute the functionality of initiator and target at full speed, and keep track of the time consumed by any modeled resources by simply incrementing a variable "on the side". Such a coding style is called *loosely-timed* in the TLM-2.0 standard. However, what the model actually does is simply to wait for the given delay on return from the **b_transport** call. This will slow down simulation because it requires a context switch per transaction, but is sufficient for the purpose of a simple example and does make the simulation log easy to interpret.

Interoperability and the Base Protocol

To summarize, TLM-2.0 ensures interoperability between models using a standard set of APIs, and provides further utility classes to enhance productivity and encourage a consistent coding style. The keys to interoperability in TLM-2.0 are:

- Use of the standard initiator and target sockets, one of which should be instantiated for each connection to a memory-mapped bus or other communication resource.
- Use of the generic payload, which should be instantiated and set to represent the attributes of each transaction object.
- Use of the base protocol.

We have not needed to explore the details of the base protocol in this particular tutorial, but it has been implicit in our use of the simple sockets and the **b_transport** method. The base protocol specifies the rules for using the generic payload with the standard TLM-2.0 interfaces and sockets. This will be elaborated in later tutorials.

The blocking transport interface should be used wherever a transaction can be completed in a single function call; effectively, the transaction request is carried with the call to **b_transport**, and the response is carried with the return from **b_transport**. In this example, a single transaction object is being reused across calls. The storage for the transaction object is allocated once-and-for-all by the initiator at the start. This is acceptable, because only one transaction is "in flight" at any one time. Memory management is trivial and is handled by the initiator.

Another feature of this particular example is that the blocking transport method does not *block*, that is, does not call wait. **b_transport** could call **wait**, however, and in principle we could have the situation where there are several concurrent calls to **b_transport** through the same socket from multiple threads in the initiator, possibly with conflicting timing annotations. This situation is permitted under the rules of the base protocol.

The blocking transport interface is designed to support the loosely-timed coding style, where the focus is on functional execution with minimal timing detail and minimal simulation overhead.

You will find the source code for this first example in file tlm2_getting_started_1.cpp.