cadence®

Interface Additions to the *e* Language for Effective Communication with SystemC TLM 2.0 Models

By Hannes Froehlich [hannes@cadence.com]

The last several years have seen strong adoption of transaction-level models using SystemC™ TLM 2.0. Those models are used for software validation and virtual prototyping. For functional verification, TLMs have a number of advantages—they are available earlier, they allow users to divide their focus on verifying functionality and protocol/timing details, they enable higher level reuse, and they can be used as reference models in advanced verification environments. Leveraging these benefits requires a convenient and seamless interface to TLM 2.0. New additions to *e* in the Cadence® Specman technology portfolio enable verification engineers to communicate efficiently with SystemC models that have TLM 2.0 interfaces.

Contents

Motivation for an *e* TLM 2.0 interface

Over the last several years there has been a very strong adoption of SystemC Transaction-Level Modeling (TLM) 2.0 for high-level modeling. Those high-level models are used in a variety of flows; for example, software development using virtual prototypes or performance analysis of processor-based SoCs.

From the perspective of functional verification, high-level models are important for several reasons. First, those models represent the functionality of a device, and as such they need to be verified. Verifying high-level models have multiple benefits:

- Verification can start earlier (high-level models often are available earlier in the design and verification cycle)
- Divide and conquer (focus on functionality during verification of high-level models and on protocol and timing details at RTL)
- Most/all of the high-level verification environment can be reused during RTL verification

Second, the high-level models are often used as reference models in advanced verification environments.

Interaction with high-level models during verification implies that the verification environment has a convenient, seamless, and efficient interface to TLM 2.0. This application note outlines the new additions to *e* that enable verification engineers to efficiently communicate with SystemC models that have TLM 2.0 interfaces.

Overview of TLM 2.0 Communication Mechanisms

The SystemC TLM2.0 standard [1] defines a set of interfaces for transporting transactions. They include blocking and non-blocking transport interfaces, a debug transport interface, and a direct memory interface. The interfaces pass transactions between initiator and target sockets. A socket gives users access to all the interfaces and enables convenient binding of the forward and backward paths.

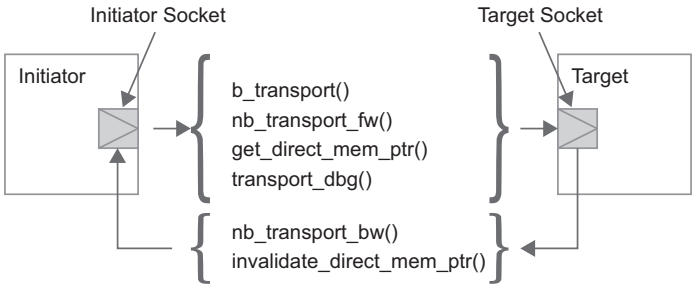

Figure 1: Initiator and target sockets with core interfaces

To increase model interoperability, a generic transaction called the tlm_generic_payload is used for transferring information between sockets. The generic payload contains control attributes (such as address, command, byte enable) and data (represented as a list of byte). This transaction payload enables abstract modeling of memory-mapped buses. The standard also defines a base protocol with which transactions are transferred across the interfaces.

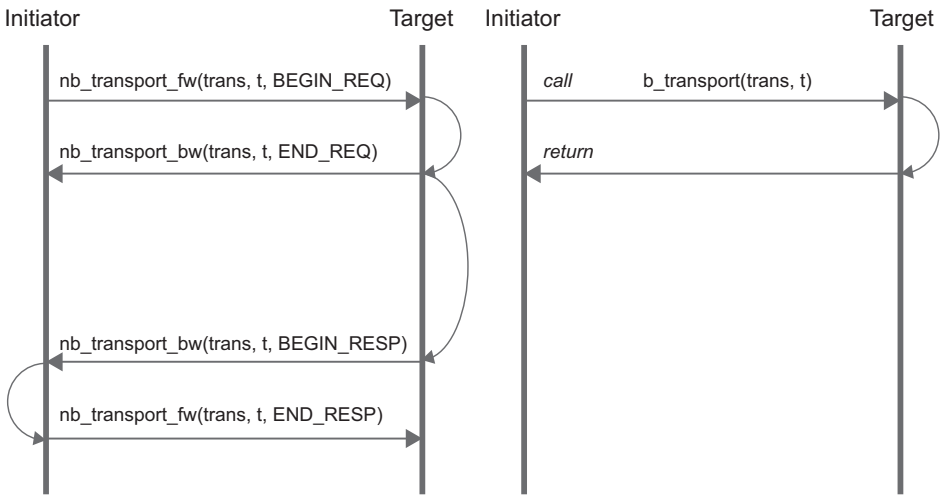

Figure 2: Non-blocking and blocking transport sequences using the base protocol

The generic payload does not include every attribute found in all memory-mapped bus protocols. Therefore, an extension mechanism is defined in the standard. It enables users to define additional attributes of a specific bus interface. The generic payload contains a list of ignorable extensions, which enables users to introduce additional attributes while keeping a high level of interoperability.

TLM 2.0 enables users to employ different modeling styles in terms of granularity of timing:

- Loosely-timed modeling: uses the blocking transport interface and offers two timing points—the beginning of a request when the blocking transport call is made, and the end of a response when the blocking transport call returns; this style is used for software development using virtual platforms
- Approximately-timed modeling: a more detailed style that uses the non-blocking transport interface; it separates the request transactions from the responses—this coding style is used for architectural exploration and performance analysis

e TLM 2.0 Interface

Introduction

The SystemC TLM 2.0 standard defines a host of functionality. For the interaction of a verification environment with TLM 2.0 models, not all of those features and utilities are required. The *e* TLM 2.0 interface focuses on socket-based communication of the generic payload, which reflects the core functionality.

Implementation of the interface requires some additional types. All newly defined types are closely aligned with the SystemC TLM 2.0 standard in terms of names and enumeration values.

All newly added types and enums are listed below (and are explained in detail in subsequent sections):

- Two new kinds of ports to represent sockets: tlm_initiator_socket and tlm_target_socket
- A new struct, tlm_generic_payload, to reflect the generic payload and a new struct tlm_extension for the extension mechanism
- Several additional enumerated types required for the interface: tlm_phase_enum, tlm_command, tlm_response_status, tlm_sync_enum, and tlm_endianness
- The singleton object ml_uvm has been extended with some utility functions

The communication between *e* and SystemC is achieved using the multi-language library of Cadence Incisive® Enterprise Simulator. This library provides the infrastructure to enable cross-language synchronization and type mapping on the language boundary. From a user's point of view, this means that you have to adopt the multi-language flow, which requires certain specific compile and runtime switches for the tools.

e sockets

To enable convenient access to SystemC TLM2.0 sockets, you can use a corresponding socket in *e*. The *e* sockets are syntactically similar to *e* ports. The syntax is:

name : [list of] *tlm-socket-type* [using *prefix-suffix*] [is instance];

- tlm-socket-type: tlm_initiator_socket | tlm_target_socket [of <t>]
- <t>: transaction type; currently supports only tlm_generic_payload

Sockets can only be instantiated under units (like ports). In addition, the user has to implement the required forward or backward interface functionality. This is done by defining the contents of the methods and time consuming methods (TCMs) that are associated with each of the sockets.

Initiators enable users to access the forward path interface functions via methods calls to the connected target sockets. Initiators need to provide the backward path interface function implementation. If you instantiate an initiator (that is, a socket with the tlm_initiator_socket type), you need to implement the nb_transport_bw() method in the unit where the socket is instantiated.

Target sockets respond to calls from the initiators by implementing the methods/TCMs called by the initiators. Specifically, a unit that instantiates a socket of type tlm_target_socket needs to implement the b_transport() TCM, the nb_transport_fw(), and the transport_dbg() methods.

Note: Failure to implement the required methods or TCMs will result in load time errors.

```
extend MASTER simple_driver {
   i_sckt : tlm_initiator_socket of tlm_generic_payload is instance;

   // implement non-blocking transport bw method of initiator
   nb_transport_bw(resp: tlm_generic_payload,
                   phase: *tlm_phase_enum, t: *time): tlm_sync_enum is {
      . . .
   };
   . . .
};
```

Figure 3: Example initiator socket instantiation and backward path implementation

To distinguish the implementations of multiple sockets in a unit, you can use the prefix or suffix attribute. The value of the attribute is added to the name of the interface methods and TCMs. For example, if you had an initiator_ socket with prefix="s1_", the non-blocking backward transport method associated with this socket would be called s1_nb_transport_bw().

Callback method registration is currently not supported. The relevant methods and TCMs must be implemented in the unit that instantiates a socket, and must be named according to the TLM 2.0 standard and the optional prefix or suffix.

An initiator creates transactions and passes them to the target by calling a core interface of the socket. This is achieved by calling methods of the initiator or the target socket, using the "socket$.<transport_method>" port notation. From an initiator socket, you can access the b_transport(), nb_transport_fw(), and transport_dbg() methods/TCMs of the connected target socket. Likewise, from a unit containing a target socket, you can access the nb_transport_bw() method of the initiator socket.

```
extend MASTER simple_driver {
   i_sckt : tlm_initiator_socket of tlm_generic_payload is instance;
   . . .
   pull_and_drive_items() @ sequencer.clock is {
      var req: tlm_generic_payload; var t: time = 0; var p:tlm_phase_enum;
      while (TRUE) {
         req = sequencer.get_next_item();
         if blocking then {
            if(i_sckt$.b_transport(req, t) != TLM_COMPLETED) {
               dut_error("MASTER simple_driver b_transport failed")};
         } else {
            p = BEGIN_REQ;
            if(i_sckt$.nb_transport_fw(req, p, t) != TLM_ACCEPTED) {
               dut_error("MASTER simple_driver nb_transport_fw failed");};
         };
         emit sequencer.item_done;
      };
   };
   . . .
};
```

Figure 4: Calling of the initiator socket interface methods

Note that some parameters in the interface methods/TCMs are reference parameters (*). This is done to enable the called method/TCM to update the values of those parameters. This means that you cannot pass scalar values, but must instead pass variables that hold the desired values—*e*.g. the time argument of b_transport() [2].

The generic payload struct

A new struct type, `tlm _ generic _ payload`, has been defined and added to the Specman® pre-defined types. All the attributes of this struct are listed below.

```
struct tlm_generic_payload like any_sequence_item {
   %m_address: uint(bits:64);
   %m_command: tlm_command;
   %m_data: list of byte;
   %m_length: uint;
   %m_response_status: tlm_response_status;
   %m_dmi: bool;
   %m_byte_enable_length: uint;
   %m_byte_enable: list of byte;
   %m_streaming_width: uint;
   %m_extensions: list of tlm_extension;
};
```

Figure 5: The tlm_generic_payload struct

Since this struct is inherited from any_sequence_item, you can use it directly in sequences.

```
extend WRITE gp_burst_seq {
   !burst: tlm_generic_payload;
   . . .
   body() @driver.clock is only {
      do burst keeping {
         it.m_command == TLM_WRITE_COMMAND;
         it.m_address == address;
         it.m_length == data_len;
         it.m_data == data_list;
         it.m_byte_enable_length == 0;
         it.m_byte_enable == {};
         it.m_streaming_width == 4;
         it.m_response_status == TLM_INCOMPLETE_RESPONSE;
      };
 . . .
```

Figure 6: Generation of generic payload content

You can also use existing sequences and add a field of type `tlm_generic_payload` to your sequence item.

To facilitate the extension mechanism of the TLM2.0 standard, a new type—`tlm_extension`—has been added to the Specman pre-defined types. This type is a struct, but has no pre-defined field members or methods/TCMs.

Users can derive a new type from tlm_extension and add attributes in the derived type. This new extension can be added procedurally to the `tlm_generic_payload` using the set_extension() method or via constraints when generating payloads in sequences

```
struct AXI_extension like tlm_extension {
   %m_burst_length: uint;
   %m_size: uint;
   %m_burst: amba_pv_burst_t;
};
. . .

extend WRITE gp_burst_seq {
   !burst: tlm_generic_payload;
   !my_AXI_extension: AXI_extension;
   body() @driver.clock is only {
      my_AXI_extension = new with {
         .m_burst_length = 16;
         .m_burst = AMBA_PV_FIXED;
         .m_size = 4;
      };
      do burst keeping {
         it.m_command == TLM_WRITE_COMMAND;
         it.m_data == data_list;
         it.m_extensions.size() == 1;
         it.m_extensions[0] == my_AXI_extension;
         . . .
```

Figure 7: Adding extensions to tlm_generic_payload

Multi-Language Registration and Communication

To enable communication between SystemC and *e*, the Incisive simulator uses a multi-language library. This library enables the following multi-language features:

- Instantiation and configuration
  - Users can instantiate and configure verification components in different languages; there are specific features that enable passing of configuration information across the language boundaries
- Synchronization of elaboration phase
  - Process of environment creation (build/construct phase) and binding of multi-language connections (connect phase)

- Passing of data across language boundaries
  - Ability to pass data objects from one language domain to another language domain; due to different memory management and organization in different languages, this process generally consists of packing and serialization of data in one domain, and de-serialization and unpacking in the other domain

The TLM 2.0 interface uses the multi-language library for synchronized build and cross-language binding, and for cross-boundary data passing. From a user's point of view, this means the following:

- SystemC TLM 2.0 sockets that are to be bound to *e* sockets need to be registered for multi-language communication; the multi-language library supplies some SystemC macros for this registration
- Users have to adopt a flow that includes irun, and they need to use one or several of the following irun command options: `-uvmtest`, `-uvmtop`, or `-ml _ uvm` (these options indicate a multi-language flow, which will automatically load the multi-language library); in addition, the switches determine in which order the build phases are called.

Registration of SystemC TLM 2.0 sockets for multi-language communication

To connect SystemC TLM 2.0 sockets to *e* sockets, the SystemC TLM 2.0 sockets need to be registered with the multi-language library. You can do this via one of the following SystemC macros:

- ML_TLM2_REGISTER_TARGET (module_i,tran_t,sckt,buswidth)
- ML_TLM2_REGISTER_INITIATOR (module_i,tran_t,sckt,buswidth)
  - module_i – module instance
  - trans_t – transaction type
  - sckt – socket instance (member of module_i)
  - buswidth – bus width
- The macros return a string that represents the full hierarchical socket name; this can then be used with the tlm2_ connect function to multi-language binding
- The macros can be called only during the construction phase (*e*.g. in the constructor of a SystemC module)
- The macros cannot be called from sc_main
- The macros are defined in ml_tlm2.h, which resides in <IES-install>/tools/uvm/uvm_lib/uvm_ml/sc/ml_uvm

```
#include "ml_tlm2.h"
. . .
class top_sc_dut : public sc_module {
public :
  simple_memory mem_0; // simple memory instance
  top_sc_dut(sc_module_name name_) : sc_module(name_), mem_0(){
    std::string full_socket_name =
    ML_TLM2_REGISTER_TARGET(mem_0, tlm_generic_payload, tsocket, 32);
    . . .
  }
};
```

Figure 8: Multi-language registration of SystemC TLM 2.0 sockets

Binding SystemC TLM 2.0 and *e* Sockets

The multi-language binding of SystemC and *e* TLM 2.0 sockets can be controlled from either language. Procedural binding mechanisms are applied at the end of the synchronized elaboration phase.

On the SystemC side, you can use the ml_tlm2_connect function. It's defined in the `ml _ tlm2.h` mentioned previously. This function should be called in the same scope as the socket registration macros.

- Void ml_tlm2_connect(std::string initiator_name,std::string target_name, bool map_transactions = [0|1])
  - Use the return value from multi-language registration macros for the SystemC sockets
  - Use the hierarchical name for *e* sockets, starting from sys
  - The argument order is significant, so put the initiator before the target

```
#include "ml_tlm2.h"
. . .
class top_sc_dut : public sc_module {
public :
  simple_memory mem_0; // simple memory instance
  top_sc_dut(sc_module_name name_) : sc_module(name_), mem_0(){
    std::string full_socket_name =
    ML_TLM2_REGISTER_TARGET(mem_0, tlm_generic_payload, tsocket, 32);
    . . .
    ml_tlm2_connect("sys.my_env.driver.i_sckt",full_socket_name);
  }
};
```

Figure 9: Multi-language binding from SystemC

To bind sockets from the *e* side, you can use the connect() pseudo-method of the socket ports. Again, the connect() pseudo-method should be called after construction, during the connect phase in *e*.

- <socket-expression>.connect("path-to-external-socket");

```
extend MASTER simple_driver {
   . . .
   connect_ports() is also {
      i_sckt.connect("top.mem_0.tsocket");
   };
};
```

Figure 10: Multi-language binding from e

Multi-Language Flow with irun

The multi-language use of TLM 2.0 sockets currently requires the use of irun. In addition to all switches required to compile and run SystemC and *e*, users must ensure the multi-language flow [3] and add some specific switches:

- Specify include paths for SystemC compilation, to enable the inclusion of the multi-language header files
- Use one of the following options to enable the multi-language flow
  - -uvmtest - declare the root entity of the logical multi-language testbench
  - -uvmtop - declare a top entity
  - -ml_uvm - enable uvm_ml

```
-I${TLM2_LAB}/my_dma/include
-I/export/home/cad/121/tools/uvm/uvm_lib/uvm_sc/sc

-I`ncroot`/tools/uvm/uvm_lib/uvm_ml/sc/ml_uvm

${TLM2_LAB}/my_dma/src/simple_dma.cpp
${TLM2_LAB}/my_dma/src/top_tb.cpp
-DSC_INCLUDE_DYNAMIC_PROCESSES
-scsynceverydelta ON
-sysc
-tlm2
-sctop top_testbench

my_dma/tests/my_dma_tb.e

-uvmtop my_dma/tests/dma_t1.e

-uvmhome `ncroot`/tools/uvm
```

Figure 11: Example irun command file

Summary

The new interface additions of Cadence Specman technology enable a convenient and efficient way to connect *e* testbenches to SystemC models with TLM 2.0 interfaces. Standard TLM 2.0 communication mechanisms are supported, which allows verification engineers to focus on verification and not have to worry about the interface.

References

[1] OSCI TLM-2.0 Language Reference Manual

[2] Incisive Enterprise Specman Elite Testbench Specman *e* Language Reference Version 12.1

[3] Incisive Verification Kits UVM Multi-Language Reference Version 12.1