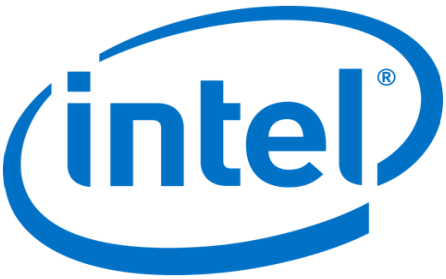

Reuse of external SystemC IPs in Intel® CoFluent™ Studio v5.1.0

An Intel® CoFluent™ Application Note

Table of Contents

1 Introduction

1.1 Aim of this document

This aim of this document is to illustrate how to reuse external SystemC IPs that exchange data through non-CoFluent ports using Binding relations in a CoFluent model.

1.2 Context

Most SystemC IPs that are developed outside of Intel CoFluent Studio do not exchange data through CoFluent ports and relations. Instead, these external SystemC IPs exchange data through *sc_in* and *sc_out* ports for Cycle-Accurate IPs or through *initiator* and *target* sockets for TLM-2 IPs.

In this application note we illustrate how to reuse external SystemC IPs in a CoFluent model. We first focus on the reuse of external SystemC IPs that exchange data through Cycle-Accurate ports. Then we focus on external SystemC IPs that exchange data through TLM-2 sockets.
We assume that the reader already knows how to import a SystemC IP from a CoFluent model as indicated in section "SystemC IP import" in the Intel CoFluent Studio User's Guide.

2 External SystemC IPs with Cycle-Accurate ports

Let us consider an external SystemC IP that exchanges data through the following Cycle-Accurate ports.

```
sc_in< bool > RST;
sc_in< bool > CLK_WR;
sc_in< bool > CLK_RD;
sc_in< sc_int<8> > DATA_TO_SYSTEM;
sc_out< sc_int<8> > DATA_FROM_SYSTEM;
```

Figure 1 illustrates the reuse of this external SystemC IP in a CoFluent model. CoFluent functions are used to send data to / read data from the external SystemC IP through Cycle-Accurate ports, and to send a reset signal and clock signals to the external SystemC IP. Binding relations are used to represent binding operations between ports.

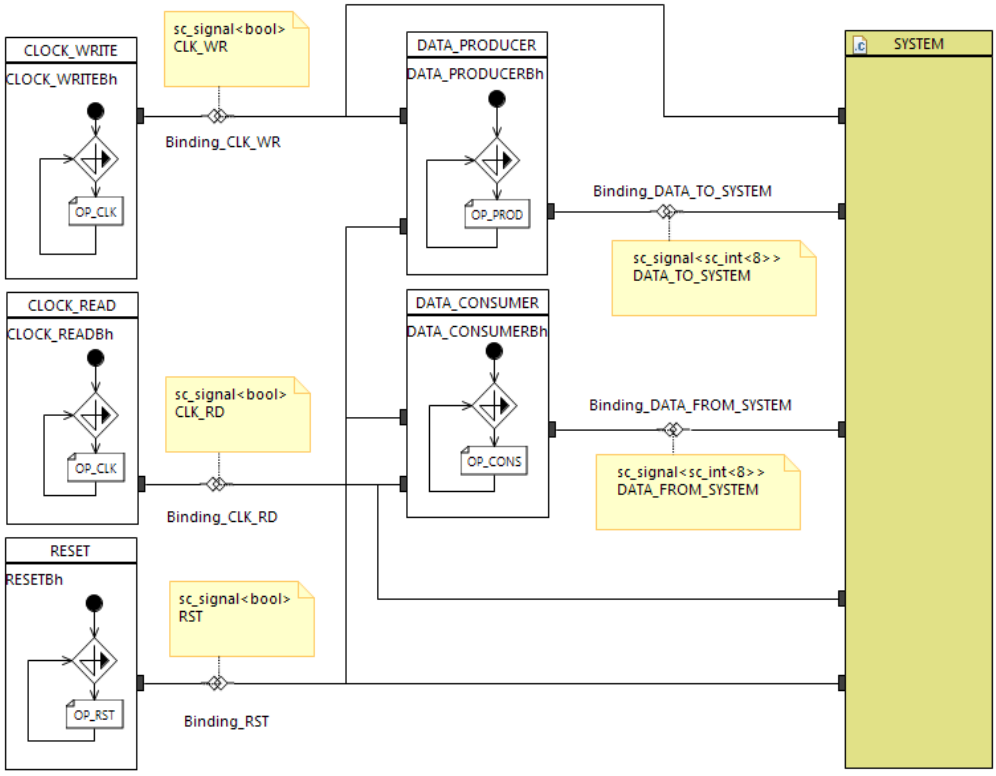

Figure 1: Re-use of an external SystemC IP with Cycle-Accurate ports in Intel CoFluent Studio v5.1.0

2.1 Cycle-Accurate ports: declaration

Cycle-Accurate ports that are used by a container function or by a leaf function must be declared in the *Declarations* section of this function. For example, Cycle-Accurate ports of the DATA_PRODUCER function are declared as follows in the *Declarations* section of this function:

```
sc_in< bool > CLK_WR;
sc_in< bool > RST;
sc_out< sc_int<8> > DATA_TO_SYSTEM;
```

2.2 Signals: declaration

Signals must be declared in the level of the hierarchy they belong to. For example, the RST, CLK_WR, CLK_RD, DATA_TO_SYSTEM and DATA_FROM_SYSTEM signals are declared in the *Declarations* section of the root structure as follows:

```
sc_signal < bool > CLK_WR;
sc_signal < bool > CLK_RD;
sc_signal < bool > RST;
sc_signal < sc_int<8> > DATA_TO_SYSTEM;
sc_signal < sc_int<8> > DATA_FROM_SYSTEM;
```

2.3 Binding relations

The binding must be done in the *Binding* section of the binding relations. For example, in the *Binding* section of the Binding_RST relation, we added the following binding between ports and signals. Ports are represented in blue and signals are represented in red.

```
RESET.RST(RST);
DATA_PRODUCER.RST(RST);
DATA_CONSUMER.RST(RST);
SYSTEM.RST(RST);
```

2.4 Sensitivity lists for leave functions

The sensitivity list of a leaf function must be specified in the *Constructor* section of this leaf function. For example, the following sensitivity list (see code in blue) has been declared in the *Constructor* section of the DATA_PRODUCER function.

Note: In the code that is generated, this line does not appear in the correct place in the constructor: it appears before the SC_THREAD declaration but the code must be modified so that it appears after as shown below. This limitation will be fixed in a future version. If you insert this line by hand between the lines that are shown in green below, then this line will not be lost when you generate again the model.

```
SC_THREAD (cfm_behavior);
//Start of user code for SC_THREAD
sensitive << CLK_WR.pos() << RST;
//End of user code
```

2.5 Algorithms of operations: write/read to/from ports

It is possible to specify graphically the behavior of CoFluent functions that send data and receive data though Cycle-Accurate ports, and to call primitives to access these ports from the algorithms of operations. Since functions are sensitive to clocks, the *Execution time* attributes of these operations must be set to 0. For example, the algorithm of the OP_PROD operation of the DATA_PRODUCER function that sends data to the SYSTEM function is shown below:

```
if(RST.read()==0){
	DATA_TO_SYSTEM.write(data_counter%128);
	…
}
wait();
```

2.6 VCD trace file generation

All signals can be monitored in a single trace file (vcd). This trace file can be declared and defined in the *Global Declarations* and *Global Definitions* sections as follows:

Global Declarations:

```
extern sc_trace_file *wf;
```

Global Definitions:

```
sc_trace_file *wf = sc_create_vcd_trace_file("VCD_FILE");
```

Then, signals are added to the trace files from the level of the hierarchy into which they are declared. For example, signals that are declared in the root of the model are added to the trace file in the *Constructor* section of the root as follows:

```
sc_trace(wf, CLK_WR, "CLK_WR");
sc_trace(wf, CLK_RD, "CLK_RD");
sc_trace(wf, RST, "RST");
sc_trace(wf, DATA_TO_SYSTEM, "DATA_TO_SYSTEM");
sc_trace(wf, DATA_FROM_SYSTEM, "DATA_FROM_SYSTEM");
```

Trace files that are created during a simulation can be opened with external tools to analyze waveforms such as GTKWave* as shown in Figure 2. Here, the DATA_MULT signal is internal to the SystemC IP.

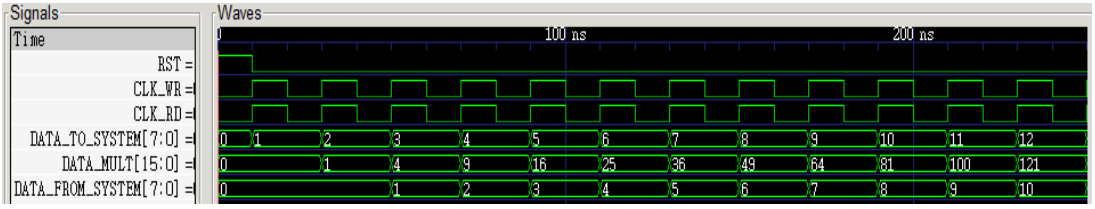

Figure 2: Waveforms

3 External SystemC IPs with TLM-2 sockets

Intel CoFluent Studio can generate wrappers automatically to ease the integration and reuse of external SystemC IPs that exchange data through TLM-2 sockets. See the "TLM 2.0 wrappers" section in the Intel CoFluent Studio User's Guide. In this application note, we propose to bypass wrappers so as to avoid unnecessary transactions that are added automatically when using wrappers.

Let us consider an external SystemC IP that exchanges data through the following TLM-2 target and initiator sockets:

```
tlm::tlm_target_socket<> target_socket_for_initiator;     // target socket
tlm::tlm_initiator_socket<> initiator_socket_for_target;  // initiator socket
```

Figure 3 illustrates the reuse of this external SystemC IP (SYSTEM) in a CoFluent model. The Initiator function sends data to the external SystemC IP through a TLM-2 initiator socket, and the Target function receives data from the external SystemC IP through a TLM-2 target socket.

In Figure 3, binding relations show how the CoFluent functions interact with the external SystemC IP through TLM-2 sockets.

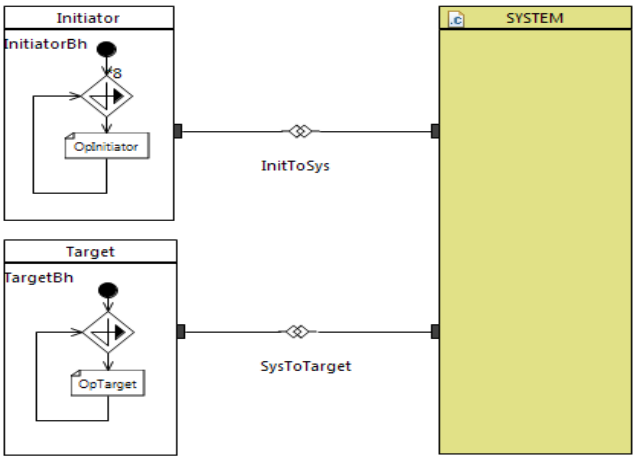

Figure 3: Reuse of an external SystemC IP with TLM-2 sockets in Intel CoFluent Studio v5.1.0

3.1 TLM-2 sockets: declaration

A TLM-2 socket that is used in a function must be declared in the *Declarations* section of this function. For example, the initiator socket of the Initiator function is declared as follows in the *Declarations* section of this function:

```
tlm_utils::simple_initiator_socket<cfm_initiator, 32> socket;
```

The target socket of the Target function is declared as follows:

```
tlm_utils::simple_target_socket<cfm_target, 32> socket;
```

3.2 Binding

The binding is done in the *Binding* section of the Binding relations. For example, we added the following code to the *Binding* section of the InitToSys Binding relation to connect the initiator socket of the Initiator function to the target socket of the SYSTEM function:

```
Initiator.socket(SYSTEM.target_socket_for_initiator);
```

We added the following code to the *Binding* section of the SysToTarget Binding relation to connect the initiator socket of the SYSTEM function to the target socket of the Target function:

```
SYSTEM.initiator_socket_for_target(Target.socket);
```

3.3 Callback methods for target sockets

A target socket and its callback method must be declared in the same *Declarations* section. For example, the callback method that is called when the target socket of the Target function receives a transaction is declared in the *Declarations* section of the Target function as follows:

```
void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& delay);
```

Then the association between a target socket and a callback method that are declared in a function must be done in the *Constructor* section of this function. For example, the association between the target socket and the callback method in the Target function is done in the *Constructor* section of this function as follows:

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

Note: "cfm_target" is the name of the class that is generated by Intel CoFluent Studio for the Target function.

The behavior of a callback method that is declared in a function must be defined in the *Definitions* section of this function. For example, the behavior of the callback method that is declared in the Target function is defined as follows in the *Definitions* section of this function:

```
void cfm_target::b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t){
    if (trans.get_command() == tlm::TLM_WRITE_COMMAND) {
      …
    }
    trans.set_response_status(tlm::TLM_OK_RESPONSE);
    sc_core::wait(t);
    ev_start.notify();     // send start event to OpTarget operation
    sc_core::wait(ev_done);// wait for done event from OpTarget operation
}
```

At the end of its execution, this callback method sends an event to start the execution of the OpTarget operation in the Target function and then waits for an event from the OpTarget operation. Thus, the behavior of an operation can be synchronized with a callback method (i.e., with the reception of a transaction by a target socket). The events must be declared in the *Declarations* section of the function into which they are used. In this example, the ev_start and ev_done events are declared as follows in the *Declarations* section of the Target function:

```
sc_core::sc_event ev_start;
sc_core::sc_event ev_done;
```

3.4 Algorithms of operations

Transactions can be sent to sockets from the algorithms of operations. For example, the following code is used in the algorithms of the OpInitiator operation in the Initiator function to send a transaction to the initiator socket:

```
trans.set_address(…);
trans.set_command(…);
…
socket->b_transport(trans, t);
```

Note: trans and t are declared in the *Declarations* section of the Initiator function as follows:

```
tlm::tlm_generic_payload trans;
sc_core::sc_time t;
```

4 Conclusion

In this application note, we illustrated how to reuse external SystemC IPs that exchange data through Cycle-Accurate ports or through TLM-2 sockets in CoFluent models using Binding relations.