Chapter 17. C++ and SystemC code generation (Experimental)

Author: Seongnam Kwon and Soonhoi Ha

In the CGC domain, C is the target language of the generated software code. C is one of the most famous and powerful high level language, but many engineers use another high level language to design, simulate, and implement systems. So someone may want PeaCE to generate software codes in another language. For this reason, we are developing two experimental targets (C++ code generation target and SystemC code generation target) in this chapter. In particular SystemC is adopted as the initial system specification language in many system level design tools. Therefore PeaCE may be used as a front-end tool to those design tools if PeaCE generates SystemC code from the Task model specification.

17.1 C++ code generation target

The C++ code generation target generates a class for each task in PeaCE. Since variables defined inside a class are only visible in the class, name conflict between blocks will be resolved much easier compared with C code generation. Moreover if there are multiple task instances of the same task in a system, they can share the class code with different instances, and this can make the binary code size much smaller.

If block should be rewritten for the C++ code generation, redundant effort and time must be needed. In order not to do that, in PeaCE, generating C++ code is automatically performed with a little modification of blocks using “CGC star parser.” This is also used in SystemC code generation target.

17.1.1 CGC Star parser

To translate a C code block into a C++ code block automatically, block structure should be explored, which is what the “CGC star parser” does. The “CGC star parser” reads and analyzes C code streams, and stores the structure of code streams in a data structure. C++ code generation target reconstructs the code streams into C++ code streams as shown in Figure 17-1. Note that this target is incomplete and has some limitation. Most critical limitation is that “#ifdef…#endif” is not supported. The current “CGC star parser” cannot analyze nested “#ifdef…#endif” structure, so ignore it. If you want to use this target, you should eliminate “#ifdef…#endif” statements inside any code stream in your block definition. And other rare coding patterns cannot be analyzed by the “CGC star parser.” This will be corrected in the following version.
17.1.2 Running C++ code generation target in Hae

At first, open the schematic “schematic/Peace/Demo/CGC/CPP/DivX/DivXPlayer_cpp”. This example consists of three tasks – AviParser, H.263 decoder, and mp3 decoder. And each task is assigned “CGC-CPPGeneration” target in the “CGC” domain as shown in Figure 17-2. The top schematic of DivX player example should be assigned the “default-BP” target in the “BP” domain as shown in Figure 17-3. Other parameters are the same as in the original DivX player example. In short, to generate a C++ code, each task should be assigned the “CGC-CPPGeneration” target and the top schematic should be assigned the “default-BP” target.
Figure 17-2 Schematic of the DivX player example - CGC-CPPGenerationTarget

Figure 17-3 Top schematic with the default-BP target
Then just click the run icon, then PeaCE generates C++ codes in the directory 
“$(HOME)/PEACE_SYSTEMS/(project_name)/func/”. In this example, codes will be generated in the directory 
“$(HOME)/PEACE_SYSTEMS/DivXPlayer_cpp/func/”. Its structure is similar to “CGC-TM” target.

17.2 SystemC code generation target

For fast HW/SW cosimulation, the transaction level modeling(TLM) is becoming important in system level 
design. SystemC is a C++-extension language developed to model the hardware and the software blocks with a 
unified language. In TLM, an hardware block is modeled in SystemC for fast simulation speed with high 
abstraction level.

In this version, PeaCE generates not the TLM simulation code but the **untimed functional simulation code**. The 
TLM simulation code will be generated in the next version. A model is generated for each task, and the top model 
is generated in the top schematic to bind the models of tasks. To communicate between models, sc_fifo is used as 
channel.

The usage of this target is same as the “CGC-CPPGeneration” target except that the systemC library should be 
installed.

17.2.1 Running SystemC code generation target in Hae

Open the schematic “schematic/Peace/Demo/CGC/SystemC/DivX/DivXPlayer_systemC” as shown in Figure 
17-4. This example is same as in the C++ code generation target, but the target of each task should be set to the 
“CGC-SystemC” target in the “CGC” domain.

Just click the “run” icon, then PeaCE generates SystemC codes. These codes will be located in the directory 
“$(HOME)/PEACE_SYSTEMS/(project_name)/func/”: in this case, this directory will be 
“$(HOME)/PEACE_SYSTEMS/DivXPlayer_systemC/func/”.

Note that the name of the makefile for generated codes is fixed as “Makefile.linux”, not the name of the project 
file, and the name of the executable binary is also fixed as “run.x”. If you want to run this code later, you should 
execute “run.x” or rename it for later use.
17.2.2 Structure of generated SystemC codes

Figure 17-4 Schematic of the DivX Player example – CGC-SystemCTarget

Figure 17-5 The structure of generated SystemC code - DivX player example

Figure 17-5 shows the structure of the generated SystemC code for the DivX player example. Each task is generated as a SystemC model. In this example, three tasks - AviParser, H.263 decoder and MP3 decoder - are generated. Communications between tasks are performed through the “sc_fifo” channel. Each task has “sc_fifo_in/out” ports to be connected to sc_fifo channels. The “Top” model initiates SystemC models of tasks.
and channels, and when simulation starts, each model runs in parallel. If one model cannot write data to another model, it is blocked until channel is available and data can be written. In the ‘read’ case, it is same as ‘write’. If one model cannot read data from another model when data is needed, this model is blocked until data can be read. If channel is too small, deadlock can occur. But it is not detected automatically. It is up to the system designer.

<table>
<thead>
<tr>
<th>AviParser.h</th>
<th>AviParser.cc</th>
</tr>
</thead>
</table>
| #include <systemc.h>
| ...                 | #include “AviParser.h”  |
| class AviParser : public sc_module {
|          private:    | ...                     |
|   int member_functions();
|   int member_var;
| public:             | // member function definitions....
|   int preinit();
|   int init();
|   int go();
|   int wrapup();
|   sc_fifo_out<char> outPort0;
|   sc_fifo_out<char> outPort1;
|   sc_fifo_out<char> *outPort[2];
|   void InitParameters()
|     { outPort[0] = &outPort0;
|     outPort[1] = &outPort1;
|     preinit();
|     init();
|   }
|   SC_CTOR(AviReader_systemCl0)
|     : outPort0("outPort0"), outPort1("outPort1")
|     {
|     AviReader_systemCl0::InitParameters();
|     SC_THREAD(go);
|     }
| int write_port(int chid, char * data, int size);
| int read_port(int chid, char * data, int size);
| int available(int chid);
| int init_port(int taskId, int portId);
| }                     | void AviParser::go()
|                       | { while(1) {
|                       |     //main codes....
|                       | }                     |
|                       | int AviParser::write_port(int chid, char * data, int size) {
|                       |     if channel doesn’t have enough buffer, wait....
|                       |     write data to channel (chid is port number)
|                       | }                     |
|                       | int AviParser::read_port(int chid, char * data, int size) {
|                       |     if channel doesn’t have enough data, wait....
|                       |     read data to channel (chid is port number)
|                       | }                     |
|                       | int AviParser::available(int chid) {
|                       |     return available data number;
|                       | }                     |
|                       | int init_port(int taskId, int portId) {
|                       |     initialize port
|                       | }                     |

Figure 17-6 Generated SystemC pseudo codes of AviParser

Figure 17-6 shows generated SystemC pseudo codes of AviParser. In the header file, output ports (outPort0, outPort1) are declared, and in function “InitParameters()”, they are mapped to the pointer array “outPort[]”. This pointer array enables the port to be accessed with index number in the generated code. All functions about port I/O (read_port, write_port…) access I/O ports with this index. Main codes of AviParser are in the function “go()” and this function runs as like SC_THREAD type in SystemC codes. For this purpose, in “AviParser.cc”, main codes are wrapped within “while(1) {….}” statements.

AviParser writes data to other tasks with “write_port()”. In function “write_port()”, if there is not enough buffer space to write into, the model is blocked until buffer is available. In contrast, in the function “read_port()”, if there are not enough data to read in the buffer, this model is blocked until data is available. Function “available()”
returns the count of available data. So if “available()” is used before “read_port()” is called, remained data can be read without blocking.

```
#include "AviParser.h"
...
class top : public sc_module
{
public:
    AviReader_systemCI0 * AviReader_systemCI0_object;
    H263Decoder_systemCI3 * H263Decoder_systemCI3_object;
    mp3_task_systemCI5 * mp3_task_systemCI5_object;
    sc_fifo<char> channel_0;
    sc_fifo<char> channel_1;

    virtual void InitParameters() { }
    virtual void InitInstances();
    virtual void DeleteInstances();

    SC_CTOR(top)
    : channel_0("channel_0", 10000), channel_1("channel_1", 10000)
    {
        top::InitParameters();
        top::InitInstances();
    }

    virtual ~top() {
        top::DeleteInstances();
    }
};

void top::InitInstances()
{
    AviReader_systemCI0_object = new AviReader_systemCI0("AviReader_systemCI0_object");
    H263Decoder_systemCI3_object = new H263Decoder_systemCI3("H263Decoder_systemCI3_object");
    mp3_task_systemCI5_object = new mp3_task_systemCI5("mp3_task_systemCI5_object");
    AviReader_systemCI0_object->outPort0(channel_0);
    AviReader_systemCI0_object->outPort1(channel_1);
    H263Decoder_systemCI3_object->inPort0(channel_0);
    mp3_task_systemCI5_object->inPort0(channel_1);
}

void top::DeleteInstances()
{
    delete all instances....
}

int sc_main(int ac, char * av[])
{
    top * top_object = new top("top_object");
    sc_start(-1);
    delete top_object;
}
```

Figure 17-7 Generated SystemC pseudo code of the top model
Figure 17-7 shows the generated SystemC pseudo code of the top model. In this model, models of tasks are declared. In the “InitInstances()” function, these models are instantiated and channels are connected to the ports. The top model is instantiated in “sc_main” and starts execution.

The detailed communication protocol is up to the SystemC library in this version. But in the next version, TLM simulation codes will be generated.