Synthesis of complex control structures from behavioral SystemC models

Francesco Bruschi, Fabrizio Ferrandi
Politecnico di Milano
Italy

Abstract

In this paper we present the results of a set of experiments we conducted in order to evaluate the viability of the behavioral synthesis, relying on the tools available at the moment in EDA market. To accomplish this we modelled a complex PCI bus interface in SystemC using a behavioral style of description. Then we tried to synthesize it by means of the Synopsis CoCentric SystemC Compiler tool. The problems arisen during synthesis, in particular those concerned with the cycle-accurate timing behavior of the synthesized circuit, were addressed. After analyzing them, possible solutions were proposed, were possible. Finally, a summary of the pros and cons of the behavioral synthesis in SystemC is presented.

1 Introduction

In order to manage the fast growth in the complexity of the digital systems, the level of abstraction adopted within the design methodologies is rising constantly. The new design languages allow descriptions of the systems at levels of abstraction not possible before. SystemC [1] [5] [6] is one of the most promising languages belonging to this category, thanks to its features oriented to the system level design.

The increased level of abstraction in the modelling must be correlated to an analogous improvement in the input specifications that can provided to the synthesis tools[2]. Automatic synthesis must be performed on systems described at a higher level than RTL in order to quicken the design cycle and let the complexity of the new designs be managed efficiently.

EDA market offers tools that allow to synthesize circuits described at the behavioral level of abstraction. Such tools are available for different traditional hardware design languages such as Verilog and VHDL.

Tool vendors are beginning to propose such synthesis tools even for the new system level design languages such as SystemC. The role of these tools in the new design flows will be of great importance, since they will allow to approach automatic synthesis after a few refinement phases from the functional specification.

In this paper we describe a set of modelling and synthesis experiences performed using SystemC as the modelling language in the design of an interface to a PCI bus.

These experiences are original in the field of the behavioral synthesis in that they address the design of circuits that have always been considered domain of the RTL description style.

We applied the behavioral style of description to the design of such a system with a twofold objective:

1. verify whether this style of description is suitable for this kind of circuits;
2. test if the behavioral synthesis tools today available allow to obtain satisfying results.

The specification of the system was composed by the PCI bus protocol on one side, and by the desired interface towards the application on the other.

2 SystemC Synthesizable subset

In this section we will briefly outline the SystemC subset allowed by the synthesis tools we considered. In the subsequent sections we will consider the effects on the design of the limitations imposed by this subset. The complete definition of the SystemC synthesizable subset allowed by SystemC Compiler can be found in [4]. More information on the behavioral modelling style in SystemC can be found in [3].

2.1 Data Types

A subset of the SystemC data types can be used in behavioral synthesizable models:

- \texttt{sc\_bit} and \texttt{sc\_bv}: represent logical bits and vectors of bits;
- \texttt{sc\_int}, \texttt{sc\_bigint}, \texttt{sc\_bigint}, \texttt{sc\_biguint}: integers types of different precision, signed and unsigned;
• `bool`: the `bool C` type;
• `C` types `int, long, short` along with their modifiers;

Note that there are no resolved types in the behavioral synthesis language subset; these can only be handled in RTL descriptions.

### 2.2 Process types

The only process type allowed for behavioral synthesis is the `sc_thread` one. This is basically a process that is sensitive only to a clock and to no other signal. In addition, a reset signal must be declared by means of the SystemC watching statement.

### 2.3 Temporal behavior and scheduling modes

The simulation semantics of the `wait` statements inside an `sc_thread` is to suspend execution until the next active clock front. From the point of view of the synthesis, these statements define the operations that have to be performed in a single clock cycle or, equivalently, they are used to define the temporal behavior of the system with cycle accuracy.

There are two basic scheduling modes that SystemC Compiler can use to synthesize a behavioral model:

- Cycle-fixed: the timing of the description is strictly preserved, that is, all blocks of instructions between two `wait` statements are scheduled to be executed in a single clock cycle, if possible;
- Superstate-fixed: the scheduler can split sequences of instructions to be executed in a single clock cycle over many cycles; the I/O semantic is preserved.

### 2.4 `wait` statements and control structures

An important set of constraints imposed to the behavioral descriptions to be synthesized are related to the position of the `wait` statements with respect to control structures such as loops and conditional statements. We will show in section 4 that most of these constraints are equivalent to the following, defined in [4]:

If one branch has a conditional (if...else, `switch...case`, or the `? ;` operator) has at least one wait statement, then place at least one wait statement in each branch (including the default branch). You can have a different number of `wait` statements in the branches.

As we shall show, this constraint rose some problems in the design of the PCI bus interface.

The rules that define the synthesizable behavioral style are called *Behavioral Coding Style Rules*, and we will refer to them as *coding rules* further on in this document.

### 3 Design Description

The PCI Local Bus is a high performance, 32-bit or 64-bit synchronous bus with multiplexed address and data lines. The bus is intended for use as an interconnect mechanism between highly integrated peripheral controller components, peripheral add-in boards, and processor/memory systems.

For our purposes, we considered an interface that can interconnect a module to the bus in both master and slave transfer modes. All the basic PCI transactions are allowed; the only set of functionalities that were not implemented are those related to the Plug’n’Play capabilities.

The interface offers to the application a simple access to the bus, hiding all the details of the complex PCI protocol.

#### 3.1 Implementation choices

One of the first problems in the modelling of a PCI Interface with the synthesizable behavioral subset of SystemC was the data types choice. The only data types used in the design are `bool` and `sc_bw`, since the resolved types don’t belong to the behavioral subset of SystemC Compiler. This was an issue, since the bus signals are intrinsically tristate.

In order to manage this problem the design was partitioned into two separate modules:

- a behavioral one that implements the main PCI-protocol interface behavior;
- an RTL one that acts as a tristate signal driver towards the PCI bus lines;

This choice was suggested by the wider set of signal types that can be handled in the RTL descriptions with respect to the behavioral ones, as for instance the resolved types such as `sc_Jv`.

The behavioral module of the interface contains two clocked thread processes: one that implements the features of the PCI bus protocol handling and the other that manages the parity errors report.

The adoption of a separate process for the error reporting was suggested by the fact that the PCI bus protocol requests particular care to the signals driving policies: the parity error signal (PERR#) used to notify errors during the data phases must be kept asserted for at least two clock cycles before being deasserted; in order to accomplish this behavior without putting to idle the rest of the transaction another process, parallel to the main one, was instantiated.
All the four types of operations performed by the interface are handled within a single process, since, with the exception of the error report functionality, the behavior could be formulated with a minimum amount of parallelism; this allowed a description of the functionality that is sequential for the great part. Moreover, this also allowed us to maintain the description "as behavioral as possible", with the aim to evaluate this style of description more effectively.

The main process must be sensitive to reset requests signaled on the bus lines, and, in order to perform a sort of error recovering, even by other processes of the interface. Since multiple reset watching isn’t allowed by the behavioral synthesizer, the RTL module was used to logically sum all the reset requests and to produce an unique signal that could be watched by the process. The interaction between the behavioral and the RTL modules are shown in picture 1.

The main process was partitioned into several functions, in order to make the code more readable and manageable.

The main function implements the basic cycle that watches the evolution of the bus and interface signals. According to the signal states detected, it invokes functions for the address decoding, chooses what operation is to be performed, and invokes the corresponding handling functions.

We found the possibility to partition the various operations into modular functions very effective during the design implementation. This is a peculiarity of the behavioral description style; in RTL an analogous possibility would be to partition the functionality into interacting processes, but that would imply a totally different semantic.

An other reason for intensively partitioning the design functionality into sub-functions was related to the possibility of efficiently applying some of the code transformations that will be described in section 4, in order to overcome some expressiveness limits imposed by the synthesizable subset allowed by the tool.

As previously mentioned, the RTL module has the task of driving the bus line signals according to the behavioral process commands and to compute the global reset signal, which is obtained from the three signals $RST\#$, from the PCI bus lines, $RESET\_INT$ from the application and $RESET$ from the behavioral module. To implement the driving functionality all the signals that are tristate have been duplicated adding the prefix ‘$z$’ to their names; so when the behavioral module has to assign ‘$Z$’ to a port for example $REQ$ what it has to do is to assert the relative port that is $ZREQ$.

Another important limit related to the behavioral synthesis is the impossibility to use bidirectional ports; this problem was also overcome with the aid of the RTL driver; in the behavioral module, each bidirectional $inout$ port was split into two different ports: an $in$ and an $out$ port (for instance: $FRAME$ was split into $FRAME\_IN$ and $FRAME\_OUT$). The RTL module receives the type of operation to be performed (reading or writing) by means of a special signal from the behavioral module, and acts correspondingly.

## 4 Synthesis issues

One of the major limits encountered in the specification of the PCI bus protocol in behavioral style was the impossibility to synthesize, with an cycle fixed scheduling mode, a set of control structures that all have in common an "unbalanced if", that is, an $if-else$ control structure with $wait$ statements in one branch and none in the other. This type of control structure violates one of the coding rules imposed by [4].

In the following section we will:

1. show some general situations in which such control structures are essential in the behavior modelling of the system;
2. explain how we tried to overcome the impossibility to express such structures;
3. try to reformulate the problem in terms of synthesis of control state machines. In a set of synthesis tests we conducted, we noted how it was impossible to generate a particular control state machine pattern, corresponding to the "unbalanced if" structure. This reformulation is of great help in understanding the problem from a different (more synthesis-oriented) point of view.

### 4.1 Exception handling

Let’s assume that, at a certain point of the execution, an error condition needs to be checked. If the error condition is true, a series of actions, possibly lasting more than one clock cycle, should be taken. Otherwise (the error condition being false), the system should go on with the nominal behavior. In the PCI bus protocol this situation arises when, for instance, the interface has to check whether a pause signal has been asserted by the device interfaced to the bus; if the pause signal is asserted, a pause handling procedure is invoked (note that we are using a "behavioral" style to describe the situation rather than a more low-level RT-level one). After "returning" from the pause handling, the system continues with its nominal behavior. Note that we don’t want to lose any clock cycle if the pause condition is false. The code from the PCI interface representing such a situation is:

```c
if (pause_in.read()==ASSERTED)
{
    do
    {
```
4.2 Signal event watching with timeout

Another situation that can be retraced to the unbalanced-if problem is the waiting of a signal event with timeout. Let us assume that the systems has to check if a signal is high. If it is not, the system should wait until it rises; at the same time, we want to count the waiting clock cycles since the system begun waiting, in order to cause a timeout if the waiting period exceeds some limit. A proper modelling of the desired behavior could be done with a for cycle:

```c
[...] for (int count=0; count<=TIMEOUT &&
    sig.read()!=ASSERTED; count++)
   wait();
```

Again, the execution of this control shouldn’t require any clock cycle in case the signal is already raised. This situation arose while modelling the PCI interface behavior with respect to some error signal assertion. Even in this case the code violates a coding rule [4]:

There should be at least one wait statement before the loop.

As we shall show further on, this problem can be reduced to the "unbalanced if" one.

4.3 Solutions proposed

The solutions proposed consist of some sort of code manipulation. We found that these manipulations allow the description of the desired behavior in a way that can be synthesized by SystemC compiler. The main drawback deriving from such solutions is that the code gets quickly very hard to understand and maintain.

The only effective solution found in order to preserve the correct timing behavior of the system was to apply some code transformation patterns to every problematic control structure. Consider the following code pattern:

```c
if (condition)
{
   f();
   wait();
}
```

This code is in contrast with the general coding rule 4, since there is no else branch in the outermost if containing a wait statement. By adding an else statement containing a single wait, as suggested in the behavioral coding style guide, we would of course alter the timing behavior of the system (it would mean losing a clock cycle each time the verification of the `pause_in` signal is performed, disregarding its value).
The previous code conflicts with a coding rule imposed in [4], since there is no else branch containing at least one wait in the if statement.

The code transformations to be applied in order to obtain a synthesizable code are constituted by the following steps:

1. add an empty else branch

2. "push" all the code between the if-else statement and the first wait() (included) into the conditional branches.

In this way, the if-else construct becomes balanced in both paths and the code is synthesizable. Almost all of the problems encountered in the behavioral level coding can be led back to the "unbalanced if" structure. For instance, in the signal event waiting with timeout, the code originally written was:

```c
for (int i=0; i<=TIMEOUT && sig.read()!=ASSERTED; i++)
    wait();
```

This code, in order to be synthesizable with cycle-fixed scheduling, lacks of a % statement immediately before the loop. We can unroll the first iteration of the loop and consequently shorten the number of iterations by one:

```c
wait(); for (int i=0+1; i<=TIMEOUT && sig.read()!=ASSERTED; i++)
    wait();
```

This piece of code is not equivalent to the first one in that the execution of the first loop body (here a simple wait()) is not guarded upon the condition sig.read()!=ASSERTED. We can then guard the execution of the "partially unrolled" loop with an if control statement in the following way:

```c
if (sig.read()!=ASSERTED) {
    wait();
    for (int i=0+1; i<=TIMEOUT && sig.read()!=ASSERTED; i++)
        wait();
}
```

Again, we are in an unbalanced if situation. We can then apply the code transformations previously described in order to get the code synthesizable.

4.4 Problem description in terms of control FSMs

In order to more formally understand the problems mentioned, we conducted a series of synthesis experiments of very simple control structures. For each one of the synthesized circuits, we analyzed the control state machine generated by SystemC compiler and compared it with the machine we would expect to be synthesized. Most of these experiments were conducted with a superstate-fixed scheduling mode. In this way, we could analyze in which way the generated circuit missed the original timing rather than just verifying that the scheduling was impossible, as would have happened with a cycle-fixed scheduling. The most useful experiment in order to understand the problem was the synthesis of the following code:

```c
#include <systemc.h>

SC_MODULE(synth_test) {
    sc_in_clk clock;
    sc_in<bool> reset;
    sc_in<bool> in;
    sc_in<bool> start;
    sc_out<bool> out;

    void while_cond_variable();

    SC_CTOR(synth_test) {
        SC_CTHREAD (while_cond_variable,clock.pos());
    }

    void synth_test::while_cond_variable() {
        // reset
        out.write(false);
        wait();
        while(true) {
            while (!start.read()) wait();
            bool cond=true;
            while (in && cond) {
                cond=false;
                wait();
            }
            out.write(true);
            wait();
        }
    }
}
```

Let us consider the second innermost while loop in the code segment: the loop condition can be true for one iteration at most. The control flow semantics of this structure is...
equivalent to that of an "unbalanced if". However, note that, while the "unbalanced if" can never be scheduled, whatever the I/O scheduling mode chosen, this structure can be scheduled in superstate-fixed mode. It is clear that the timing of this description implies that the outermost while loop could be executed in two clock cycles; that is, in the control state machine we should find a cycle of length two in the subdiagram representing the outermost while loop. The control state machine generated by SystemC Compiler is the one in figure 2.

Figure 2. Control FSM generated

As it can be seen, there is no path of length two in the subdiagram corresponding to the main loop. In order to manually "fix" the control state machine to obtain the desired behavior, it would suffice to add an edge from the state labelled "A" to the state labelled "C", with the same actions associated with the transition from "B" to "C". We didn't find any way to let SystemC Compiler generate such a state machine pattern.

5 Conclusions

In this paper we showed some modelling and synthesis experiences using the behavioral level of abstraction in SystemC. The system designed is a PCI bus interface that let an application connect to such a bus in a simple way. The specification was complex enough to let us evaluate the advantages of adopting behavioral synthesis tools in our design flow.

Using a behavioral level of abstraction allowed us to start from a functional description of the system in SystemC, validate it, and then refining the module to be synthesized in a smooth way. The transition from the functional to the behavioral level can be managed in an efficient way, and was performed in a relatively brief period.

On the other hand, the main drawbacks of using the behavioral synthesis tools nowadays available are:

- the constraints imposed to the synthesizable subset limit to the possibility of precisely expressing the desired timing behavior; the solutions we found to overcome these limitations have a high cost in terms of code readability and maintainability;
- there is no support for the handling of typical hardware data types such as the resolved signals; this imposes the use of RTL descriptions even when there is no real need of them;

These turned out to be major issues in the development of a control intensive application such as the PCI bus interface. We think these problems will have to be addressed in order to let behavioral synthesis be suitable for a significant set of applications.

References