How to Model a FIR Filter in SystemC?

In the following, we will use SystemC to model a simple Finite Impulse Response (FIR) filter. The block diagram of a 5th order FIR filter is given in Figure 1. We use the filter which is used as an example in the laboratory exercise for the Methods and Algorithms for System Design course (ET4054) given at TUDelft.

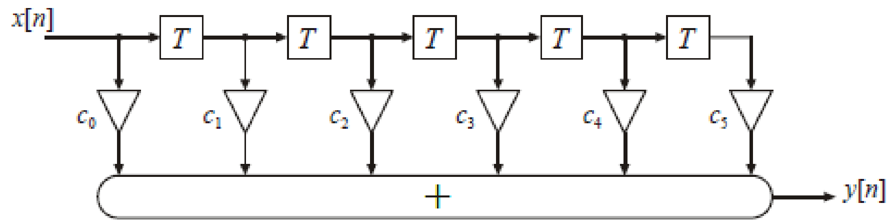

Figure 1. Block diagram of a 5th order FIR filter.

This filter is made up of five delay elements that are all clocked with the sample frequency. The output of each delay element is individually weighted with a coefficient, and is combined with the others in a weighted sum.

For the coefficients given in Table 1, this filter will show a low-pass transfer function in the frequency domain as shown in Figure 2.

Table 1. Coefficients for the 5th order FIR filter.

Coefficient	Value
c_0	-0.07556556070608
c_1	0.09129209297815
c_2	0.47697917208036
c_3	0.47697917208036
c_4	0.09129209297815
c_5	-0.07556556070608

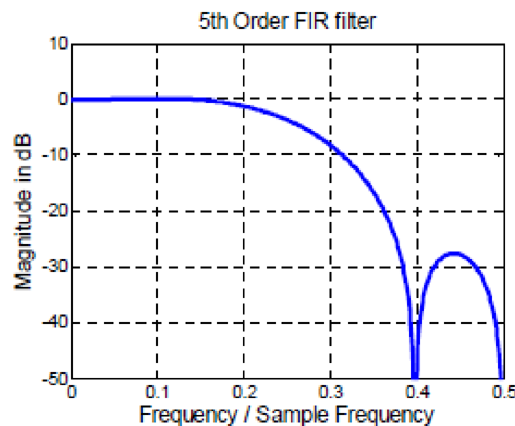

Figure 2. Magnitude transfer function of the 5th order FIR filter.

TUDelft has developed a scheduling toolbox which can be used to implement DSP algorithms like this FIR filter in VHDL [Scheduling Toolbox for MATLAB, Reference Guide]. Several scheduling algorithms can be explored using this tool. The tool can also perform a design space exploration using the list scheduling algorithm and can minimize the latency by applying a retiming algorithm. The tool uses an input file to describe the DSP algorithm called a cir-file (short for circuit file). The cir-file is a simple ASCII text file with the extension ".cir", in which basically all operations to be performed are written line by line. The cir-file is just a textual representation of the sequencing graph (SG) of the DSP algorithm. The SG of the FIR filter from Figure 1 is given in Figure 3. The delay elements used in the DSP algorithm are also explicitly given in the cir-file. The cir-file for the 5th order FIR filter is given in Figure 4.

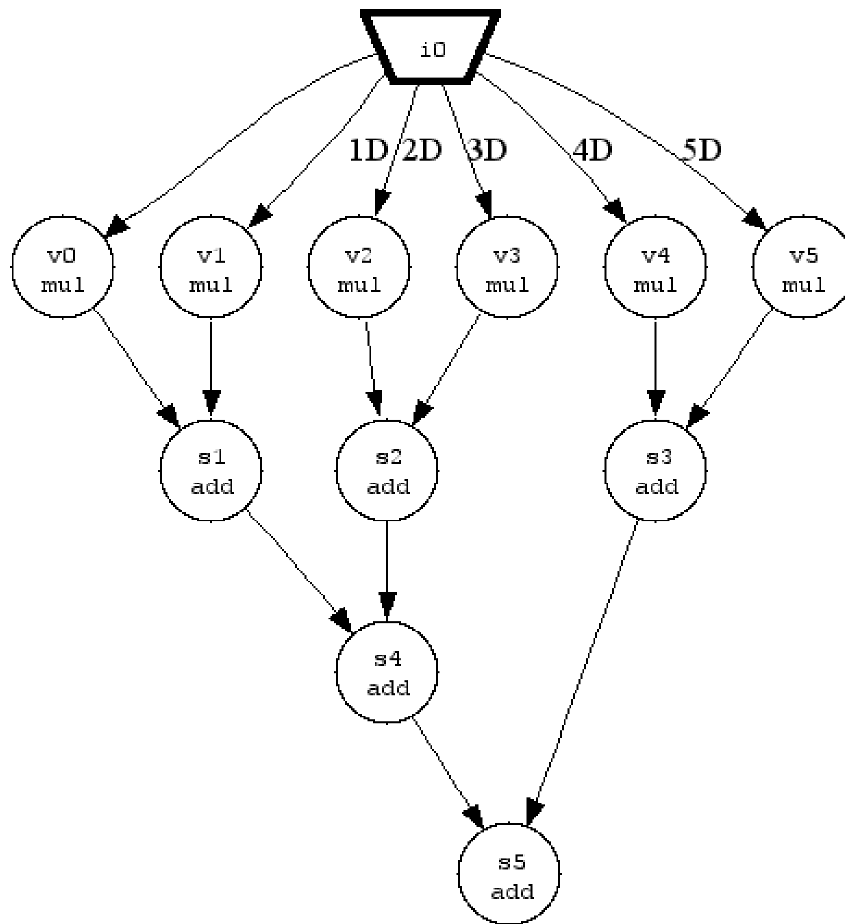

Figure 3. The sequencing graph for a 5th order FIR filter.

```
% operations
v0 = c0 * i0;
v1 = c1 * i1;
v2 = c2 * i2;
v3 = c3 * i3;
v4 = c4 * i4;
v5 = c5 * i5;
s1 = v0 + v1;
s2 = v2 + v3;
s3 = v4 + v5;
s4 = s1 + s2;
s5 = s4 + s3;
% output node
o5 = s5;
% cross connections
o0 = i0;
o1 = i1;
o2 = i2;
o3 = i3;
o4 = i4;
% feedback through delay elements
i1 = To0;
i2 = To1;
i3 = To2;
i4 = To3;
i5 = To4;
```

Figure 4. Cir-file for the 5th order FIR filter.

All characters following the character `%` on a line in the cir-file are considered to be comments to make the cir-file better readable for humans. The scheduling tool discards the `%` character and all characters following it up to the end of the line. An identifier starting with the character `i` represents an input port, an identifier starting with `o` represents an output port, and an identifier starting with `c` (or `a`) represents a (constant) coefficient. Each line in a cir-file which is not an empty line or a comment must contain one assignment. Such assignment should have one of the following forms:

- *`identifier = identifier operator identifier`*
  This assignment form represents an operator node in the SG. Only the `*`, `-`, and `+` operators are allowed. The name of the node as displayed by the tool equals the identifier used on the left hand side of the assignment. On the right hand side of the assignment input ports and previous described operator nodes can be used.
- *`output port = identifier`*
  This assignment form represents the connection to an output port. Each output port should be explicitly connected to (only one) operator node or input port.
- `input port = Toutput port`
  This assignment form represents a delay element.

Each assignment can optionally be followed by a `;` character. The cir-file has some more advanced features which are not relevant for this example.

The values of the coefficients are defined together with the values of the input signal(s) in a file called an .INP file. This file is read by the VHDL test bench which is generated by the Scheduling Tool. In our SystemC model we have chosen to use fixed coefficients for the FIR filter.

Because we want to transform our SystemC description of the FIR filter into a Data Flow Graph (DFG) which can be implemented by tools like the Scheduling Toolbox we start with a SystemC model which resembles the format used in the cir-file. The FIR filter will be modeled as a SystemC module with a clock input port, one data input port and one data output port. Other input and output ports like reset, start, done and error are not modeled in the behavioral SystemC model and we assume that these signals and their obvious behavior are added by the tool. The Scheduling Toolbox is capable of doing this. First the code for the SystemC test bench is given.

Test Bench

The FIR filter is modeled as a SystemC module with a clock input, one data input (`sample`) and one data output (`result`). A test bench component is used to test the FIR filter. The connections between the clock generator, the test bench, and the FIR module are shown in Figure 5.

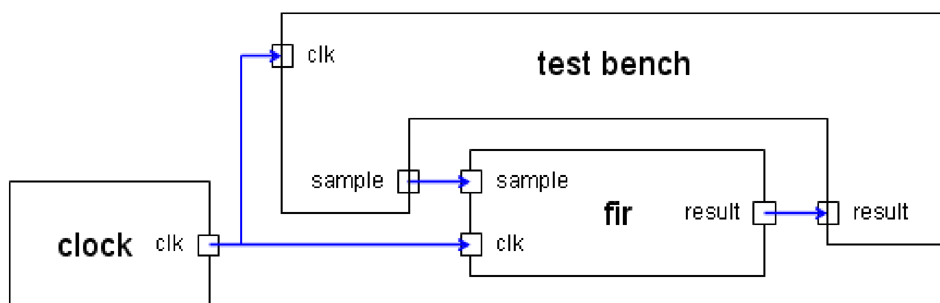

Figure 5. Test bench with FIR filter.

The test bench generates a sample on each active clock edge. It also reads the result on every active clock edge. The test bench first generates an unit impulse function and after the response is finished generates an unit step function. The expected response values for the unit impulse function and the expected response values for the unit step function are known inside the test bench. These expected values are compared to the actual response values produced by the FIR filter. The code for the test bench can be found in Appendix A.

FIR Filter SystemC Models

The FIR filter can be modeled in SystemC in several different ways. There are many choices to make:

- Each SG operator node can be used in a separate assignment statement or several operators can be combined in a compound expression.
- The delay elements can be modeled explicitly, as `sc_buffer` objects, or implicitly.
- The read or write operations to the ports and channels can be explicitly programmed or can be implicit by using the convenience type conversion and assignment operators provided by SystemC.
- The behavior of the FIR filter can be described in a `SC_METHOD` or alternatively in a `SC_CTHREAD`. We do not consider `SC_THREAD` because `SC_THREAD` is not part of the synthesizable subset of SystemC.
- The variables which are needed can be defined as separate variables or can be packed into an array.
- The coefficients can be defined inside the expressions as magic numbers or can be defined as symbolic constants.
- The FIR module can be divided in several sub modules or not.
- The channel connected to the input or output port can be read or written via the interface implemented by the channel or the input and output ports can be read or written via the read or write convenience functions provided by the ports.
- The data type of the data input port and data output port can be floating point or fixed point. Although floating point numbers are not supported in the synthesizable subset of SystemC we want to support them for verification purposes.

These choices are all independent of one another so there are at least $2^9 = 512$ different models of the same FIR filter to consider. Some of these models are given below.

FIR Filter Model 1: Based on Cir-file.

This first SystemC model of the FIR filter resembles the cir-file as close as possible. The behavior is described in a `SC_METHOD`, the delay elements are modeled explicitly by using `sc_buffer` objects. It is also possible to model the delay elements by using `sc_signal` objects but `sc_buffer` is preferred because the name buffer exactly describes the functionality of the delay element. The floating point type `double` is used as the type of the data input and data output signals. This makes it very easy to verify the results, for example by using MATLAB. The coefficients are defined inside the expressions and each operator is used in a separate assignment statement. The read and write convenience functions are used explicitly to read and write the ports. The delay elements and the variables are all defined as separate object and are not packed into an array. The SystemC code is shown in Figure 5.

```
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_CTOR(FIR) {
            SC_METHOD(behavior);
            sensitive << clk.neg();
      }
private:
      sc_buffer<double> i1, i2, i3, i4, i5;
      void behavior() {
            double i0 = sample.read();

            double v0 = -0.07556556070608 * i0;
            double v1 =  0.09129209297815 * i1.read();
            double v2 =  0.47697917208036 * i2.read();
            double v3 =  0.47697917208036 * i3.read();
            double v4 =  0.09129209297815 * i4.read();
            double v5 = -0.07556556070608 * i5.read();

            double s1 = v0 + v1;
            double s2 = v2 + v3;
```

```
            double s3 = v4 + v5;
            double s4 = s1 + s2;
            double s5 = s4 + s3;

            result.write(s5);

            i1.write(i0);
            i2.write(i1.read());
            i3.write(i2.read());
            i4.write(i3.read());
            i5.write(i4.read());
        }
};
```

Figure 6. SystemC model for 5th order FIR filter based on cir-file.

The most interesting part of the model given in Figure 5 is the modeling of the delay elements. These elements are modeled as `sc_buffer` objects `i1`, `i2`, `i3`, `i4`, and `i5`. It is important to understand that the order of the `write` operations to these buffers is irrelevant. The `behavior` method is called by the SystemC simulator on the negative edge of the `clk` signal, as specified in the sensitivity list of the `SC_METHOD`. The `write` operations to the buffers are not performed immediately but are postponed until the `behavior` method returns to the SystemC simulator. The simulator then performs these `write` operations in the next delta cycle. This behavior is identical to the concurrent assignment operator `<=` used for signals in VHDL. There is no need to initialize the `sc_buffer` objects to zero because the constructor of `sc_buffer` will take care of that [SystemC standard, p 109].

FIR Filter Model 2: Compound Expressions.

This model is a simple abstraction of model 1. All multiplications and additions are combined in one expression. The variables `i0`, `v0`, `v1`, …, `v5`, `s1`, `s2`, …, `s5` are no longer needed. This model is shown in Figure 7. The lines which are exactly the same as in Figure 6 are shown in a gray color.

```
SC_MODULE(FIR) {
    sc_in_clk clk;
    sc_in<double> sample;
    sc_out<double> result;
    SC_CTOR(FIR) {
        SC_METHOD(behavior);
        sensitive << clk.neg();
    }
private:
    sc_buffer<double> i1, i2, i3, i4, i5;
    void behavior() {
        result.write(
            -0.07556556070608 * sample.read() +
             0.09129209297815 * i1.read() +
             0.47697917208036 * i2.read() +
             0.47697917208036 * i3.read() +
             0.09129209297815 * i4.read() +
            -0.07556556070608 * i5.read()
        );

        i1.write(sample.read());
        i2.write(i1.read()); i3.write(i2.read());
        i4.write(i3.read()); i5.write(i4.read());
    }
};
```

Figure 7. SystemC model for the 5th order FIR filter using a compound expression.

Please note that the SG for model 1 is not exactly the same as the SG for model 2. The SG for model 1 is given in Figure 3 and the SG for model 2 is given in Figure 8.

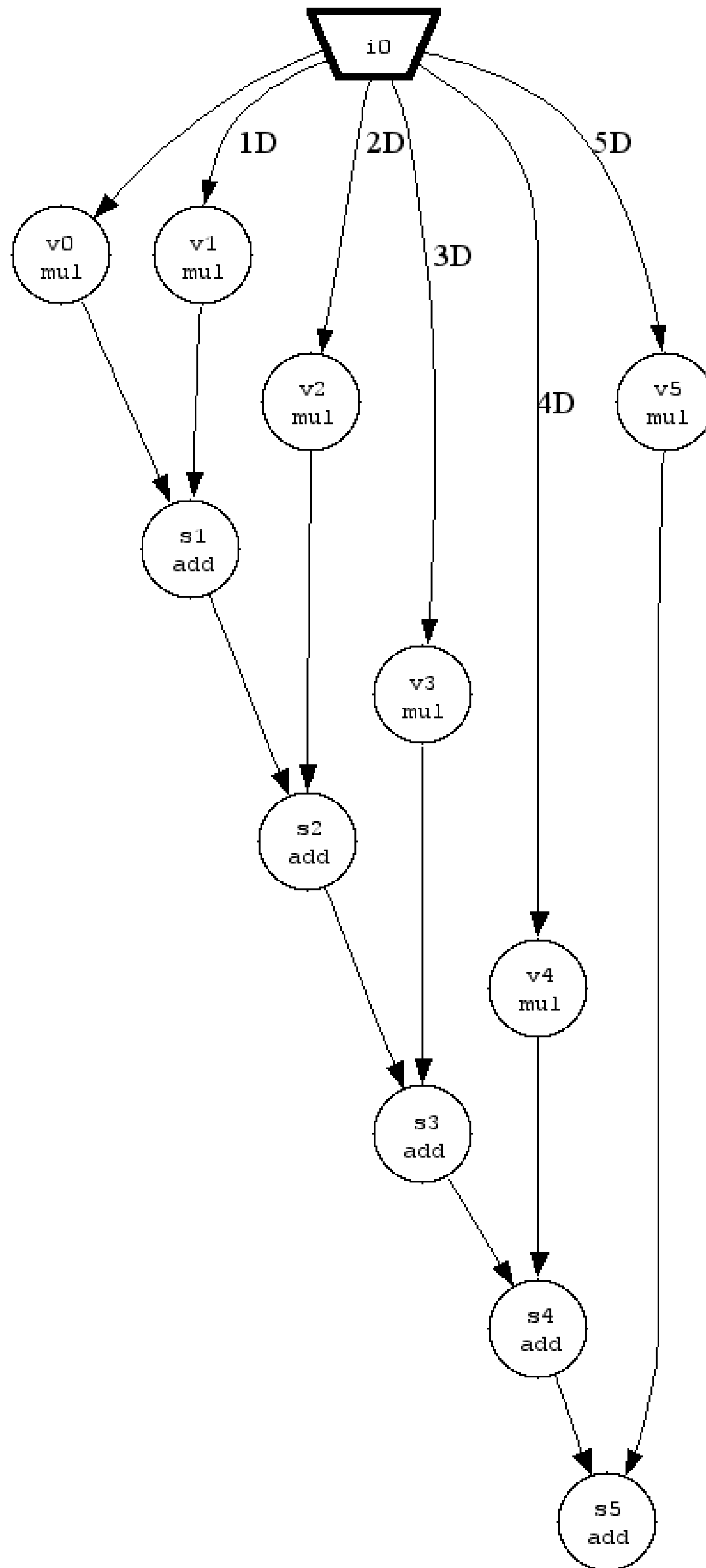

Figure 8. The sequencing graph for the 5th order FIR filter modeled in Figure 7.

As can be seen in Figure 3, the critical path between the inputs and the output consists of 1 multiplication and 3 additions. The critical path for model 2, shown in Figure 8, consists of 1 multiplication and 5 additions. This longer critical path can be explained by the way C++ evaluates expressions. The multiplication operator has a higher precedence than the addition operator and therefore the multiplications are performed before the additions. The

additions, which all have the same precedence, have to be preformed from left to right according to the C++ standard. It is possible that the tool used to implement the SG is capable of optimizing the critical path by reordering the operation nodes. The Scheduling Tool developed at TUDelft is not capable of performing this optimization. We can optimize the SG by using parentheses in the expression as shown in Figure 9.

```
result.write(
      ( -0.07556556070608 * sample.read() +
         0.09129209297815 * i1.read() ) +
      (  0.47697917208036 * i2.read() +
         0.47697917208036 * i3.read() ) +
      (  0.09129209297815 * i4.read() +
        -0.07556556070608 * i5.read() )
);
```

Figure 9. Extra parentheses added to the expression in SystemC model 2.

The SG for Figure 9 is given in Figure 3.

FIR Filter Model 3: Implicit Delay Elements.

In this model the delay elements are not modeled as `sc_buffer` objects but as normal C++ data members. This model is shown in Figure 10. The lines which are exactly the same as in Figure 7 are shown in a gray color.

```
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_CTOR(FIR): i1(0), i2(0), i3(0), i4(0), i5(0) {
            SC_METHOD(behavior);
            sensitive << clk.neg();
      }
private:
      double i1, i2, i3, i4, i5;
      void behavior() {
            result.write(
                  -0.07556556070608 * sample.read() +
                   0.09129209297815 * i1 +
                   0.47697917208036 * i2 +
                   0.47697917208036 * i3 +
                   0.09129209297815 * i4 +
                  -0.07556556070608 * i5
            );

            i5 = i4;
            i4 = i3;
            i3 = i2;
            i2 = i1;
            i1 = sample.read();
      }
};
```

Figure 10. SystemC model for the 5th order FIR filter without explicit delay elements (`sc_buffer`).

The most interesting part of the model given in Figure 10 is the modeling of the delay elements. These elements are modeled as private data members `i1`, `i2`, `i3`, `i4`, and `i5`. It is important to understand that the order of the assignment operations to these data members is of crucial importance. The assignments to these variables are performed immediately as can be expected from normal C++ variables. The data members keep their value between successive calls of the `behavior` function. When the assignment statements are preformed sequential in

this specific order the data members together behave like a shift register consisting of 5 delay elements. This behavior is identical to the sequential assignment operator `:=` used for variables in VHDL. The data members have to be explicitly initialized to zero in the initialization list of the constructor. This model is more abstract (i.e. further away from the concrete implementation) because in a hardware implementation the delay elements will operate in parallel as modeled in Figure 7.

FIR Filter Model 4: Implicit `read` and `write` Function Calls.

In all models presented so far, the `read` and `write` functions of the ports and signals are explicitly called. SystemC provides overloaded type conversion operators and overloaded assignment operators which can be used to implicitly call the `read` and `write` functions. The model shown in Figure 11 is the same as model 2, see Figure 7, but uses implicit calls instead of explicit calls to the `read` and `write` functions.

```
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_CTOR(FIR) {
            SC_METHOD(behavior);
            sensitive << clk.neg();
      }
private:
      sc_buffer<double> i1, i2, i3, i4, i5;
      void behavior() {
            result =
                  -0.07556556070608 * sample +
                   0.09129209297815 * i1 +
                   0.47697917208036 * i2 +
                   0.47697917208036 * i3 +
                   0.09129209297815 * i4 +
                  -0.07556556070608 * i5;

            i1 = sample;
            i2 = i1;
            i3 = i2;
            i4 = i3;
            i5 = i4;
      }
};
```

Figure 11. SystemC model for the 5th order FIR filter using implicit `read` and `write` function calls.

It is now very easy to misinterpret the `behavior` function. This function seems to consist of 6 sequential assignment statements. If this was the case then at the end of the function the values of all objects `i1`, `i2`, `i3`, `i4`, and `i5` would be equal to the value of `sample` and the module would not act like a FIR filter. A careful inspection of the code reveals that each assignment performs an implicit call to a `write` function and therefore all assignment statements are in fact concurrent assignment statements. These concurrent statements are not performed immediately but are postponed until the `behavior` method returns to the SystemC simulator. The simulator then performs these `write` operations in the next delta cycle.

FIR Filter Model 5: `SC_CTHREAD` instead of `SC_METHOD`.

In all models presented so far, the behavior of the FIR filter is modeled as a `SC_METHOD`. The model shown in Figure 12 is the same as model 3, see Figure 10, but uses an `SC_CTHREAD` to model the behavior of the FIR filter.

```
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_CTOR(FIR) {
            SC_CTHREAD(behavior, clk.neg());
      }
private:
      void behavior() {
            double i1=0, i2=0, i3=0, i4=0, i5=0;
            while (1) {
                  wait();
                  result.write(
                        -0.07556556070608 * sample.read() +
                         0.09129209297815 * i1 +
                         0.47697917208036 * i2 +
                         0.47697917208036 * i3 +
                         0.09129209297815 * i4 +
                        -0.07556556070608 * i5
                  );

                  i5 = i4;
                  i4 = i3;
                  i3 = i2;
                  i2 = i1;
                  i1 = sample.read();
            }
      }
};
```

Figure 12. SystemC model for the 5th order FIR filter using a `SC_CTHREAD`.

In the model shown in Figure 10 the `behavior` member function is called on every negative edge of the `clk` signal. In the model shown in Figure 12 the `behavior` member function is called only once at the start of the simulation. The wait for the next negative edge of the `clk` signal must be explicitly modeled by calling the `wait()` function. The data members `i1`, `i2`, `i3`, `i4`, and `i5` used in Figure 10 are no longer needed in Figure 12. In Figure 12 these variables are moved into the `behavior` member function. They can be modeled as local variables now because the `behavior` member function will never finish, due to the `while (1)` infinite loop. These local variables are initialized before the infinite loop, this models the initial reset behavior of the FIR filter.

The model of Figure 12 is at a lower abstraction than the model of Figure 10. In Figure 10 there is no scheduling information presented in the model. In Figure 12 the schedule can be made explicit by adding more `wait` statements. In Figure 13 a schedule is given which only uses two multipliers and two adders. The SystemC model of this scheduled FIR filter is shown in Figure 14. This FIR filter reads a sample in state 1 and produces a result in state 5. The test bench needs to be adapted to only provide a new sample, and read a result, after every 5 clock cycles.

The scheduled FIR filter can also be modeled using a `SC_METHOD` but this model will be more complex because we will need an extra data member to remember the current state.

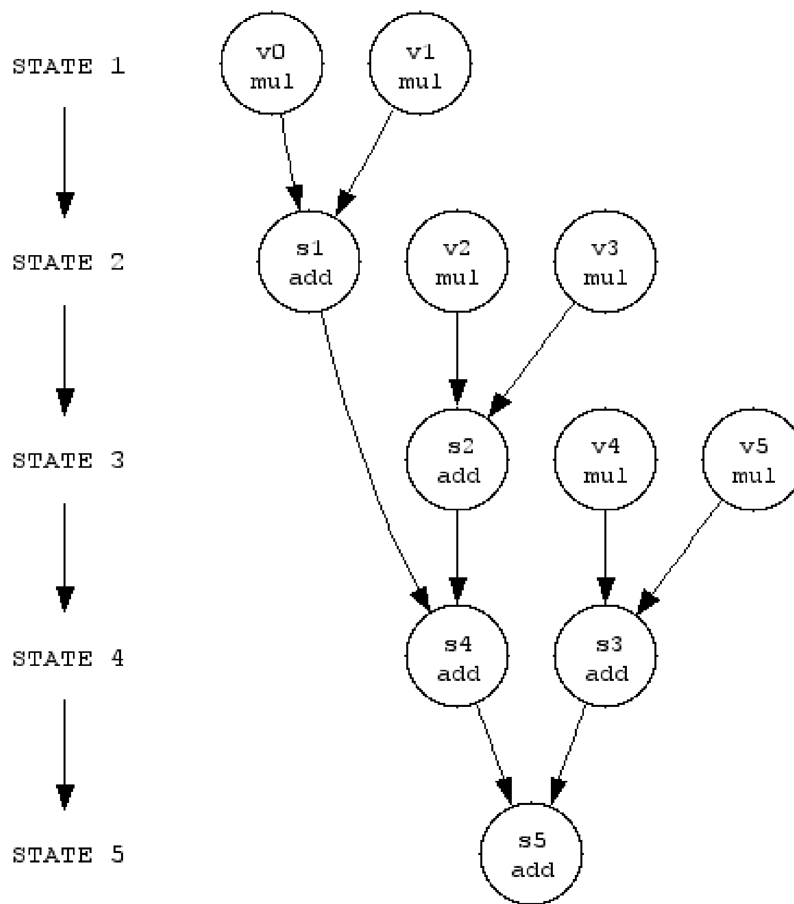

Figure 13. A schedule for the 5th order FIR filter using 2 multipliers and 2 adders.

```
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_CTOR(FIR) {
            SC_CTHREAD(behavior, clk.neg());
      }
private:
      void behavior() {
            double i1=0, i2=0, i3=0, i4=0, i5=0;
            while (1) {
                  wait();
                  double i0 = sample.read();
                  double v0 = -0.07556556070608 * i0;
                  double v1 =  0.09129209297815 * i1;
                  wait();
                  double s1 = v0 + v1;
                  double v2 =  0.47697917208036 * i2;
                  double v3 =  0.47697917208036 * i3;
                  wait();
                  double s2 = v2 + v3;
                  double v4 =  0.09129209297815 * i4;
                  double v5 = -0.07556556070608 * i5;
                  wait();
                  double s3 = v4 + v5;
                  double s4 = s1 + s2;
                  wait();
                  double s5 = s3 + s4;
                  result.write(s5);
```

```
                i5 = i4;
                i4 = i3;
                i3 = i2;
                i2 = i1;
                i1 = i0;
            }
        }
};
```

Figure 14. The SystemC model for the scheduled 5th order FIR filter using 2 multipliers and 2 adders.

FIR Filter Model 6: Symbolic Constants.

In all models presented so far, the FIR filter's coefficients are modeled as literal constants. The model shown in Figure 15 uses symbolic constants.

```
SC_MODULE(FIR) {
    sc_in_clk clk;
    sc_in<double> sample;
    sc_out<double> result;
    SC_CTOR(FIR) {
        SC_CTHREAD(behavior, clk.neg());
    }
private:
    void behavior() {
        const double c0 = -0.07556556070608;
        const double c1 =  0.09129209297815;
        const double c2 =  0.47697917208036;
        const double c3 =  0.47697917208036;
        const double c4 =  0.09129209297815;
        const double c5 = -0.07556556070608;
        double i1=0, i2=0, i3=0, i4=0, i5=0;
        while (1) {
            wait();
            result.write(
                c0*sample.read() + c1*i1 + c2*i2 + c3*i3 +
                                                c4*i4 + c5*i5
            );

            i5 = i4;
            i4 = i3;
            i3 = i2;
            i2 = i1;
            i1 = sample.read();
        }
    }
};
```

Figure 15. The SystemC model for the 5th order FIR filter using local symbolic constants.

The use of local constants within the `behavior` function is only sensible when a `SC_CTHREAD` is used to model the behavior of the FIR filter. The symbolic constants are initialized before the infinite loop and are therefore only initialized once. If, on the other hand, an `SC_METHOD` is used to model the behavior of the FIR filter this method is called on every active edge of the clock and it is seems inappropriate to initialize the symbolic constants each time the method is called. In this case the symbolic constants must be modeled outside the behavior member function. To prevent pollution of the global namespace these symbolic constants must be placed inside the `SC_MODULE` which models the FIR filter. There are two possible ways to do this: the coefficients can be modeled as constant data members of the `FIR` class or, alternatively, the coefficients can be modeled as static constant data members.

Static constants are shared by all instances of the `FIR` class, they must be initialized outside the class. Non-static constants are instantiated for each instance of the `FIR` class and must therefore be initialized in the constructor. The use of static constant data members is shown in Figure 16 and the use of non-static constant data members is shown in Figure 17. The delay elements are explicitly modeled as `sc_buffer` objects in Figure 17. This is not relevant for this example but we will use it in the next paragraph.

```
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_CTOR(FIR) {
            SC_METHOD(behavior);
            sensitive << clk.neg();
      }
private:
      static const double c0, c1, c2, c3, c4, c5;
      sc_buffer<double> i1, i2, i3, i4, i5;
      void behavior() {
            result.write(
                  c0 * sample.read() +
                  c1 * i1.read() +
                  c2 * i2.read() +
                  c3 * i3.read() +
                  c4 * i4.read() +
                  c5 * i5.read()
            );

            i1.write(sample.read());
            i2.write(i1.read()); i3.write(i2.read());
            i4.write(i3.read()); i5.write(i4.read());
      }
};

const double FIR::c0 = -0.07556556070608;
const double FIR::c1 =  0.09129209297815;
const double FIR::c2 =  0.47697917208036;
const double FIR::c3 =  0.47697917208036;
const double FIR::c4 =  0.09129209297815;
const double FIR::c5 = -0.07556556070608;
```

Figure 16. The SystemC model for the 5th order FIR filter using static symbolic constants.

```
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_CTOR(FIR):
                  c0(-0.07556556070608), c1( 0.09129209297815),
                  c2( 0.47697917208036), c3( 0.47697917208036),
                  c4( 0.09129209297815), c5(-0.07556556070608) {
            SC_METHOD(behavior);
            sensitive << clk.neg();
      }
private:
      const double c0, c1, c2, c3, c4, c5;
// See Figure 16 for the rest of this class
};
```

Figure 17. The SystemC model for the 5th order FIR filter using non-static symbolic constants.

FIR Filter Model 7: Compound data types.

In all models presented so far, all constants and variables are modeled as separate objects. It is possible to organize these objects in compound data types. The array type is in this case the most appropriate compound type to use. Figure 18 shows a SystemC model of the FIR filter which is based on Figure 15 but uses arrays instead of separate constants and variables.

```
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_CTOR(FIR) {
            SC_CTHREAD(behavior, clk.neg());
      }
private:
      void behavior() {
            const int ORDER = 5;
            const double c[ORDER+1]={
                  -0.07556556070608,  0.09129209297815,
                   0.47697917208036,  0.47697917208036,
                   0.09129209297815, -0.07556556070608
            };
            double i[ORDER+1]={0};
            while (1) {
                  wait();
                  i[0] = sample.read();
                  double sum = 0;
                  for (int j=0; j<=ORDER; ++j)
                        sum = sum + c[j] * i[j];
                  result.write(sum);
                  for (int j=ORDER; j!=0; --j)
                        i[j]=i[j-1];
            }
      }
};
```

Figure 18. The SystemC model for the 5th order FIR filter using arrays.

Please note that the SystemC model of the FIR filter given in Figure 18 is more complicated than the model given in Figure 15. An extra variable `sum` is needed to accumulate the result. In Figure 15 it is obvious that the six multiplications are independent of each other and can be performed in parallel. In Figure 18, on the other hand, it is not immediately obvious that the iterations of the first `for` loop are independent of each other. Each iteration seems to depend on the value of `sum` which is calculated in the former iteration. But if the `for` loop is unrolled the expression used in Figure 15 reappears which reveals that all multiplications can be performed in parallel. The order of the first `for` loop can be changed without any consequences but the order of the second `for` loop should not be changed. This may not be immediately obvious to some one who reads the model.

Figure 19 shows a SystemC model of the FIR filter which is based on Figure 16 but uses arrays instead of separate constants and variables. In figure 19 the delay elements are explicitly modeled using `sc_buffer` objects. In this case the order of the first `for` loop and also the order of the second `for` loop can be changed without any consequences. All `write` operations in the second `for` loop can be performed in parallel.

```
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_CTOR(FIR) {
            SC_METHOD(behavior);
```

```
            sensitive << clk.neg();
      }
private:
      static const int ORDER = 5;
      static const double c[ORDER+1];
      sc_buffer<double> i[ORDER];
      void behavior() {
            double sum = c[0] * sample.read();
            for (int j=1; j<=ORDER; ++j)
                  sum = sum + c[j] * i[j-1].read();
            result.write(sum);

            i[0].write(sample.read());
            for (int j=1; j<ORDER; ++j)
                  i[j].write(i[j-1].read());
      }
};

const double FIR::c[] = {
      -0.07556556070608,  0.09129209297815,  0.47697917208036,
       0.47697917208036,  0.09129209297815, -0.07556556070608
};
```

Figure 19. The SystemC model for the 5th order FIR filter using arrays and explicit delay elements.

The advantage of the SystemC models given in Figure 18 and 19 compared to the SystemC models given in Figure 15 and 16 is the fact that the order of the modeled FIR filter can be changed more easily. To make this more obvious a compile time symbolic constant `ORDER` is declared in both models given in Figure 18 and 19. But changing the definition of this constant is not enough to change the order of the FIR filter. A FIR filter of a higher order also needs more coefficients. Therefore it is needed to parameterize the model. Inside the FIR model C-style arrays are used. The size of these arrays depends on the order of the FIR filter. The size of a C-style array must be a compile time constant. Therefore the order must be passed to the model as a compile time parameter (e.g. template parameter). The coefficients must also be passed to the model if the model is parameterized with the order of the FIR filter. This can be done during the elaboration phase of the execution of the SystemC model via a constructor parameter or during the compilation of the SystemC model using another template parameter. Figure 20 shows a compile time parameterized version of the FIR filter given in Figure 19. Figure 20 also shows how a 5th order FIR filter can be instantiated using this template.

```
template <int ORDER, const double* c>
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_CTOR(FIR) {
            SC_METHOD(behavior);
            sensitive << clk.neg();
      }
private:
      sc_buffer<double> i[ORDER];
      void behavior() {
            double sum = c[0] * sample.read();
            for (int j=1; j<=ORDER; ++j)
                  sum = sum + c[j] * i[j-1].read();
            result.write(sum);

            i[0].write(sample.read());
            for (int j=1; j<ORDER; ++j)
                  i[j].write(i[j-1].read());
      }
};
```

```
const int ORDER = 5;
extern const double c[ORDER+1] = {
      -0.07556556070608,  0.09129209297815,  0.47697917208036,
       0.47697917208036,  0.09129209297815, -0.07556556070608
};


int sc_main (int argc , char *argv[]) {
      //...
      FIR<ORDER, c> fir("fir");
      //...
}
```

Figure 20. The SystemC model for the `ORDER`th order FIR filter using arrays and explicit delay elements.

Sometimes the order of the FIR filter can only be determined during run time. For example, because the coefficients are read from an input file and the number of coefficients read from the file should determine the order of the filter. In this case it is impossible to use C-style arrays in the FIR model because the sizes of these arrays are not known at compile time. This can be solved by using dynamic arrays. Because the values of the parameters are not known at compile time a template can not be used anymore. Therefore the parameters are passed to the constructor of the FIR module at compile time in this case. Figure 21 shows a run time parameterized version of the FIR filter given in Figure 19. Figure 21 also shows how a 5th order FIR filter can be instantiated by reading the file with coefficients shown in Figure 22.

```
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_HAS_PROCESS(FIR);
      FIR(sc_module_name name, const vector<double>& coeff):
                  sc_module(name),
                  ORDER(coeff.size()-1),
                  c(new double[ORDER+1]),
                  i(new sc_buffer<double>[ORDER]) {
            copy(coeff.begin(), coeff.end(), c);
            SC_METHOD(behavior);
            sensitive << clk.neg();
      }
      ~FIR() {
            delete[] c;
            delete[] i;
      }
private:
      const int ORDER;
      double* c;
      sc_buffer<double>* i;
      void behavior() {
            double sum = c[0] * sample.read();
            for (int j=1; j<=ORDER; ++j)
                  sum = sum + c[j] * i[j-1].read();
            result.write(sum);

            i[0].write(sample.read());
            for (int j=1; j<ORDER; ++j)
                  i[j].write(i[j-1].read());
      }
};

int sc_main (int argc , char *argv[]) {
      //...
```

```
        ifstream coefficentsFile("coeff.txt");
        vector<double> coeff;
        copy(istream_iterator<double>(coefficentsFile),
                          istream_iterator<double>(), back_inserter(coeff));
        FIR fir("fir", coeff);
        //...
}
```

Figure 21. The SystemC model for the `ORDER`th order FIR filter using dynamic arrays.

```
-0.07556556070608 0.09129209297815  0.47697917208036
 0.47697917208036 0.09129209297815 -0.07556556070608
```

Figure 22. Contents of the file coeff.txt used in Figure 21.

The model given in Figure 21 uses dynamic arrays. According to the SystemC Synthesizable Subset 1.3 [OSCI] the `new` and `delete` operators are not supported for synthesis. The model given in Figure 21 could nevertheless be synthesized, because the `new` operator is only used during the elaboration phase of the execution of the model. After the elaboration phase the sizes of the arrays do not change anymore and the process description of the model does not use any `new` or `delete` operators. If a synthesizer would execute the model until the end of the elaboration phase before synthesizes then it should be feasible to synthesize this model.

FIR Filter Model 8: Structural description.

In all models presented so far, a behavioral description of the FIR filter's functionality is given as a process description (`SC_METHOD` or `SC_CTHREAD`). Alternatively, it is possible to describe the structure of the FIR filter instead of its behavior. In figure 23 three different building blocks are described: an `S` module models an adder, a `V` module models a multiplication with a constant coefficient, and a `D` module models a delay element. These three building blocks are used to construct a model of the 5th order FIR filter. Please note that the `SC_MODULE` which models the FIR does not have any process (`SC_METHOD` or `SC_CTHREAD`) to describe its behavior. The functionality of the FIR filter is completely described by its structure. For a schematic entry tool it would be easier to produce the SystemC model given in Figure 22 for the schema given in Figure 1 than one of the behavioral models given before.

```
template<typename T>
SC_MODULE(S) {
        sc_in<T> in1, in2;
        sc_out<T> out;
        SC_CTOR(S) {
                SC_METHOD(behavior);
                sensitive << in1 << in2;
        }
private:
        void behavior() {
                out.write(in1.read() + in2.read());
        }
};

template<typename T>
class V: public sc_module {
public:
        sc_in<T> in;
        sc_out<T> out;
        V(const sc_module_name& name, const T& cc):
                                                      sc_module(name), c(cc) {
                SC_METHOD(behavior);
                sensitive << in;
```

```
	}
private:
	void behavior() {
		out.write(c * in.read());
	}
	const T c;
	SC_HAS_PROCESS(V);
};

template<typename T>
SC_MODULE(D) {
	sc_in_clk clk;
	sc_in<T> in;
	sc_out<T> out;
	SC_CTOR(D) {
		SC_METHOD(behavior);
		sensitive << clk.neg();
	}
private:
	void behavior() {
		out.write(in.read());
	}
};

SC_MODULE(FIR) {
	sc_in_clk clk;
	sc_in<double> sample;
	sc_out<double> result;
	SC_CTOR(FIR):
			v0("v0", -0.07556556070608),
			v1("v1",  0.09129209297815),
			v2("v2",  0.47697917208036),
			v3("v3",  0.47697917208036),
			v4("v4",  0.09129209297815),
			v5("v5", -0.07556556070608),
			s1("s1"), s2("s2"), s3("s3"), s4("s4"), s5("s5"),
			i1("i1"), i2("i2"), i3("i3"), i4("i4"), i5("i5") {
		v0.in(sample); v0.out(v0out);
		v1.in(i1out);  v1.out(v1out);
		v2.in(i2out);  v2.out(v2out);
		v3.in(i3out);  v3.out(v3out);
		v4.in(i4out);  v4.out(v4out);
		v5.in(i5out);  v5.out(v5out);
		s1.in1(v0out); s1.in2(v1out); s1.out(s1out);
		s2.in1(v2out); s2.in2(v3out); s2.out(s2out);
		s3.in1(v4out); s3.in2(v5out); s3.out(s3out);
		s4.in1(s1out); s4.in2(s2out); s4.out(s4out);
		s5.in1(s4out); s5.in2(s3out); s5.out(result);
		i1.clk(clk);   i1.in(sample); i1.out(i1out);
		i2.clk(clk);   i2.in(i1out);  i2.out(i2out);
		i3.clk(clk);   i3.in(i2out);  i3.out(i3out);
		i4.clk(clk);   i4.in(i3out);  i4.out(i4out);
		i5.clk(clk);   i5.in(i4out);  i5.out(i5out);
	}
private:
	V<double> v0, v1, v2, v3, v4, v5;
	S<double> s1, s2, s3, s4, s5;
	D<double> i1, i2, i3, i4, i5;
	sc_signal<double> v0out, v1out, v2out, v3out, v4out, v5out,
	                         s1out, s2out, s3out, s4out,
```

```
                                     i1out, i2out, i3out, i4out, i5out;
};
```

Figure 23. Structural SystemC model for 5th order FIR filter.

FIR Filter Model 9: Without the convenience functions provided by the ports.

In all models presented so far, the input and output ports are read or written via the `read` or `write` convenience functions provided by the ports. The SystemC Synthesizable Subset 1.3 [OSCI] states:

> *Ports represent the externally visible interface to a module and are used to transfer data into and out of the module. Ports can be declared using `sc_in`, `sc_out` and `sc_inout` constructs.*

In SystemC `sc_in`, `sc_out`, and `sc_inout` are specialized port classes which can be used with signals. For example, a `sc_in` port is derived from `sc_port<sc_signal_in_if<T>, 1>`. The `sc_signal_in_if<T>` is an interface which is implemented in several SystemC channels including `sc_signal`. This interface declaration includes a pure virtual `read` member function.

In SystemC a port is bound to a channel. This channel must implement the interface which is required by the port. A bounded port forwards interface member function calls to the channel to which that port is bound. So a general port does not know which member functions are declared in the interface it requires. To forward the member function calls a port provides an overloaded `operator->` which returns a pointer to the interface to which the port is bound. When, for example, a port `p` is bound to a channel which provides an interface which contains a `read` function the proper way to call this function through the port is `p->read()`. The overloaded `operator->` of the port returns a pointer to the interface to which the port is bound. Then the `operator->` is applied again to this interface pointer and the `read` function which is declared in the interface and implemented in the channel is called.

Because an `sc_in` port knows the interface it requires (`sc_signal_in_if<T>`) this specialized port provides convenience functions which can be used to call the functions declared in the interface. For example, `sc_in` provides a `read` member function which just calls the `read` member function of the interface to which the port is bound. Therefore, an `sc_in` port `p` can not only be read by using the general syntax `p->read()` but also be the specialized syntax `p.read()`. In figure 24 a SystemC model of the FIR filter is given which does not use the convenience functions provided by the ports.

```
SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<double> sample;
      sc_out<double> result;
      SC_CTOR(FIR) {
            SC_METHOD(behavior);
            sensitive << clk.neg();
      }
private:
      sc_buffer<double> i1, i2, i3, i4, i5;
      void behavior() {
            result->write(
                  -0.07556556070608 * sample->read() +
                   0.09129209297815 * i1.read() +
                   0.47697917208036 * i2.read() +
                   0.47697917208036 * i3.read() +
                   0.09129209297815 * i4.read() +
                  -0.07556556070608 * i5.read()
            );

            i1.write(sample->read());
            i2.write(i1.read());
```

```
            i3.write(i2.read());
            i4.write(i3.read());
            i5.write(i4.read());
        }
};
```

Figure 24. The SystemC model for the 5th order FIR filter without using the convenience functions provided by the ports.

It is also possible to model a FIR filter with ports that require another interface. In Figure 25 a `Test_interface` is declared. This `Test_interface` is implemented in the `Test_FIR` class. The data input port and the data output port of the FIR filter both require a `Test_interface`. These ports can therefore be bound to an instance of the `Test_FIR` class. This model is not in conformance with the SystemC Synthesizable Subset 1.3 [OSCI] definition because this definition only allows the use of the specialized `sc_in`, `sc_out`, and `sc_inout` ports.

```
class Test_interface: virtual public sc_interface {
public:
    virtual double get() const = 0;
    virtual void put(const double&) = 0;
};

class Test_FIR: public Test_interface {
public:
    virtual double get() const {
        // produce a sample
    }
    virtual void put(const double& result) {
        // check the result
    }
};

SC_MODULE(FIR) {
    sc_in_clk clk;
    sc_port<Test_interface> sample;
    sc_port<Test_interface> result;
    SC_CTOR(FIR) {
        SC_METHOD(behavior);
        sensitive << clk.neg();
    }
private:
    sc_buffer<double> i1, i2, i3, i4, i5;
    void behavior() {
        double i0(sample->get());
        result->put(
            -0.07556556070608 * i0 +
             0.09129209297815 * i1.read() +
             0.47697917208036 * i2.read() +
             0.47697917208036 * i3.read() +
             0.09129209297815 * i4.read() +
            -0.07556556070608 * i5.read()
        );

        i1.write(i0);
        i2.write(i1.read());
        i3.write(i2.read());
        i4.write(i3.read());
        i5.write(i4.read());
    }
};
```

```
int sc_main (int argc , char *argv[]) {
      // ...
      Test_FIR testBench;
      FIR fir("fir");
      fir.sample(testBench);
      fir.result(testBench);
      // ...
}
```

Figure 25. The SystemC model for the 5th order FIR filter using generic ports.

FIR Filter Model 10: `sc_fixed` data type.

So far, the C++ `double` data type is used for all variables used in the SystemC models of the FIR filter. Because the C++ data type `double` is not supported by the SystemC Synthesizable Subset 1.3 [OSCI] none of the FIR models presented so far are synthesizable. Instead of the C++ floating point data type `double` the SystemC fixed point data type `sc_fixed` should be used in a synthesizable model. The model shown in Figure 26 is almost the same as the model shown in Figure 7, but uses the `sc_fixed` data type instead of the `double` data type.

```
typedef sc_fixed<17, 2, SC_RND> fixed_type;

SC_MODULE(FIR) {
      sc_in_clk clk;
      sc_in<fixed_type> sample;
      sc_out<fixed_type> result;
      SC_CTOR(FIR) {
            SC_METHOD(behavior);
            sensitive << clk.neg();
      }
private:
      sc_buffer<fixed_type> i1, i2, i3, i4, i5;
      void behavior() {
            result.write(
                  -0.075566 * sample.read() +
                   0.091292 * i1.read() +
                   0.476979 * i2.read() +
                   0.476979 * i3.read() +
                   0.091292 * i4.read() +
                  -0.075566 * i5.read()
            );

            i1.write(sample.read());
            i2.write(i1.read());
            i3.write(i2.read());
            i4.write(i3.read());
            i5.write(i4.read());
      }
};
```

Figure 26. The SystemC model for the 5th order FIR filter using the `sc_fixed` data type.

Please note that the coefficients are specified in Figure 26 with less precision than in Figure 7 because the fixed point type only uses 15 binary digits after the binary point. These literal constants are converted to the required fixed point type implicitly by the compiler. The SystemC `sc_fixed` data type provides overloaded operators. The overloaded `operator*` and `operator+` are used in the description of the behavior of the FIR filter.

Appendix A. Code for the test bench and the main program used to test the FIR filter:

```
#include <systemc>
#include <iostream>
#include <iomanip>
using namespace sc_core;
using namespace std;

static int error = 0;

#define ASSERT_DOUBLES_EQUAL(expected, actual, delta) \
      assert_doubles_equal(expected, actual, delta, __FILE__, __LINE__);

void assert_doubles_equal(double expected, double actual, double delta,
                                                string file, int line) {
      if ( fabs(actual - expected) >= delta ) {
            ++error;
            cerr<< "Error in " << file << " at line " << line <<endl;
            cerr<< "    " << setprecision(15) << actual
                << " was expected to be "
                << setprecision(15) << expected << " plus or minus "
                << setprecision(15) << delta <<endl;
      }
}

SC_MODULE(Test_FIR) {
      sc_in_clk clk;
      sc_out<double> sample;
      sc_in<double> result;
      SC_CTOR(Test_FIR) {
            SC_THREAD(behavior);
            sensitive << clk.neg();
      }
private:
      void behavior() {
            double samples[16]= {
                  // impulse:
                  1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                  // step:
                  1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
            };
            double results[16]= {
                  // impulse response:
                   0.0,                -0.07556556070608,  0.09129209297815,
                   0.47697917208036,  0.47697917208036,  0.09129209297815,
                  -0.07556556070608,  0.0,
                  // step response:
                   0.0,                -0.07556556070608,  0.01572653227208,
                   0.49270570435244,  0.96968487643279,  1.06097696941095,
                   0.98541140870487,  0.98541140870487
            };
            sample.write(0);
            wait();
            int number_of_samples(sizeof samples/sizeof samples[0]);
            for (int i(0); i<number_of_samples; ++i) {
                  sample.write(samples[i]);
                  wait();
                  ASSERT_DOUBLES_EQUAL(results[i], result.read(), 2E-14);
            }
            sc_stop();
```

```
        }
};

SC_MODULE(FIR) {
// Many different implementations possible
};

int sc_main (int argc , char *argv[]) {
        sc_clock clock("clock", 10, SC_NS);
        sc_signal<double> sample;
        sc_signal<double> result;

        Test_FIR testBench("testBench");
        testBench.clk(clock.signal());
        testBench.sample(sample);
        testBench.result(result);

        FIR fir("fir");
        fir.clk(clock.signal());
        fir.sample(sample);
        fir.result(result);

        sc_trace_file *tf(sc_create_vcd_trace_file("trace"));
        tf->set_time_unit(1, SC_NS);
        sc_trace(tf, clock.signal(), "clk");
        sc_trace(tf, sample, "sample");
        sc_trace(tf, result, "result");

        sc_start();

        sc_close_vcd_trace_file(tf);

        cerr<< endl << "There " << (error!=1?"were ":"was ") << error
            << " error" << (error!=1?"s ":" ") << "found running this test!";

        cin.get();
        return error;
}
```