Application Examples

Wolfgang Haid, Kai Huang, and Simon Künzli
Revision 1069, July 23, 2008

Introduction

In this document, eight example applications are described to show how real applications can be programmed using the proposed process network Schema along with the corresponding C/C++ source code. In these examples, dataflow-oriented applications are considered.

- As an introductory example, in Example 1 a sequence of numbers is squared.
- In Example 2, iterators are used to generate a chain of processes that square a number. Using this chain of square processes, a higher power of the input number sequence is computed.
- In Example 3, iterators are used in two-dimensions. Using these iterators, a 2D mesh is generated in which tokens are passed forward in horizontal and vertical direction.
- In Example 4, the multiplication of square matrices is considered. In this example, the single matrix coefficients need to be distributed to more than one process which requires multiple <port>, <sw_channel>, and <connection> elements for each matrix coefficient.
- In Example 5, the fast Fourier transform is implemented. In the process network, the <function> element is used for generating a butterfly network. Moreover, the example shows how processes can be parametrized using the init() function. In contrast to the previous examples, non-elementary data types — in this example complex numbers — are passed between processes.
- In Example 6, a first order infinite impulse response (IIR) filter is implemented. In that example, a feedback loop is implemented.
- In Example 7, an N-th order IIR filter with N feedback loops is implemented.

- In Example 8, non-blocking functions for checking the status of communication channels are introduced.

Remark: Only Example 1 is suited to execute on the real or virtual SHAPES platform, because the other examples demonstrate advanced concepts that are currently not supported by the HdS generation.

1 Example 1

In Example 1, a canonical producer-consumer application is built, in which the producer generates data and the consumer consumes the generated data. Between the producer consumer pair, one process is inserted to process the data.

Figure 1: Example 1.

The task graph is shown in Figure 1. In this graph, there are three processes, i.e. one *generator* process, one *consumer* process, and one *square* process, with which two FIFO channels are interconnected. At each invocation, the *generator* process generates a real number with type double and transfers it to the *square* process by Channel $C1$. The *square* process gets the number, squares it, and sends the result to the *consumer* process by Channel $C2$. The *consumer* receives the final result and displays it to on the console.
The XML code to model this application is shown below:

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <processnetwork name="example 1">
03   <process name="generator">
04     <port type="output" name="1"/>
05     <source type="c" location="generator.c"/>
06   </process>
07   <process name="consumer">
08     <port type="input" name="1"/>
09     <source type="c" location="consumer.c"/>
10   </process>
11   <process name="square">
12     <port type="input" name="1"/>
13     <port type="output" name="2"/>
14     <source type="c" location="square.c"/>
15   </process>
16
```

```
17   <sw_channel type="fifo" size="10" name="C1">
18     <port type="input" name="0"/>
19     <port type="output" name="1"/>
20   </sw_channel>
21   <sw_channel type="fifo" size="10" name="C2">
22     <port type="input" name="0"/>
23     <port type="output" name="1"/>
24   </sw_channel>
25
26   <connection name="g-c">
27     <origin name="generator">
28       <port name="1"/>
29     </origin>
30     <target name="C1">
31       <port name="0"/>
32     </target>
33   </connection>
34   <connection name="c-c">
35     <origin name="C2">
36       <port name="1"/>
37     </origin>
38     <target name="consumer">
39       <port name="1"/>
40     </target>
41   </connection>
42   <connection name="s-c">
43     <origin name="square">
44       <port name="2"/>
45     </origin>
46     <target name="C2">
47       <port name="0"/>
48     </target>
49   </connection>
50   <connection name="c-s">
51     <origin name="C1">
52       <port name="1"/>
53     </origin>
54     <target name="square">
55       <port name="1"/>
56     </target>
57   </connection>
58 </processnetwork>
```

2 Example 2

The Example 2 is an extension of Example 1. The difference is that an iterator is used to generate multiple copies of the *square* process and FIFO channels connected in between. In this case, the *square* process is iterated three times, while all the copies implement the same functionality, i.e., the C source code is the same for all *square* processes.

The task graph is shown in Figure 2. In this graph, there are a total of five processes and four FIFO channels. The *generator* and *consumer* pro-

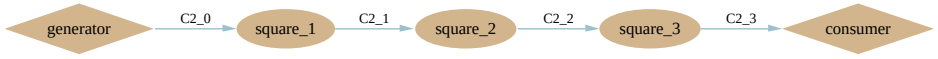

Figure 2: Example 2.

cesses are the same as those in Example 1. The set of *square_X* processes is generated using a one-dimensional iterator, which iterates three times. The channels, which connect these iterated processes, are also generated using an iterator. The iteration bounds are defined by the variable N which is set to 3. The iterator related XML code is shown below:

```
<variable value="3" name="N"/>

<iterator variable="i" range="N">
  <process name="square">
    <append function="i"/>
    <port type="input" name="0"/>
    <port type="output" name="1"/>
    <source type="c" location="square.c"/>
  </process>
</iterator>

<iterator variable="i" range="N + 1">
  <sw_channel type="fifo" size="10" name="C2">
    <append function="i"/>
    <port type="input" name="0"/>
    <port type="output" name="1"/>
  </sw_channel>
</iterator>
```

3 Example 3

In this example, a 2D mesh of *forward* processes is generated as the center part of the network. Two sets of producers, i.e. *horizontal_generator* and *vertical_generator*, generate the input of the mesh. Two sets of consumers, i.e. *horizontal_consumer* and *vertical_consumer*, get the output from the mesh.

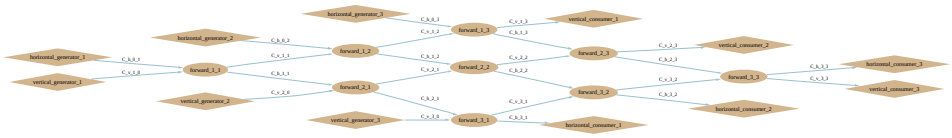

Figure 3: Example 3.

The task graph is shown in Figure 3. In this graph, a 3×3 mesh of

forward processes is generated in the center. Each *forward* process has four ports, which connect to its horizontal and vertical neighbors. This set of processes is generated using iterators in two dimensions, thus the generated processes share the same C source file. In this example, all the *forward* processes get data from horizontal and vertical inputs and directly pass them to the outputs.
The XML code to generate the *forward* processes along with the channels between the processes is shown below:

```
01 <iterator variable="i" range="N">
02   <iterator  variable="j" range="N">
03     <process name="forward">
04       <append function="i"/>
05       <append function="j"/>
06       <port type="input" name="west"/>
07       <port type="input" name="north"/>
08       <port type="output" name="east"/>
09       <port type="output" name="south"/>
10       <source type="c" location="square.c"/>
11     </process>
12     <sw_channel type="fifo" size="10" name="C_h">
13       <append function="i"/>
14       <append function="j"/>
15       <port type="input" name="in"/>
16       <port type="output" name="out"/>
17     </sw_channel>
18     <sw_channel type="fifo" size="10" name="C_v">
19       <append function="i"/>
20       <append function="j"/>
21       <port type="input" name="in"/>
22       <port type="output" name="out"/>
23     </sw_channel>
24   </iterator>
25 </iterator>
```

One-dimensional iterators are used to generate two sets of generators and two sets of consumers. In this example, the horizontal generators will put one character of the string "nopqrstuvwxyz" to the mesh at each invocation, and the vertical generators will input the string "abcdefghijklm" to the mesh in the same manner. The two sets of consumers receive the corresponding outputs and display them to standard output.

4 Example 4

As a next example application, the $N \times N$ matrix multiplication is considered. The computation of the matrix product is split up into single multiplications and additions. This way, the application "matrix multiplication" is split up into single processes, each of which only performs a multiplication

followed by an addition (see Fig. 4).

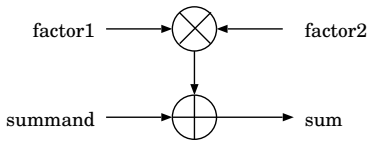

Figure 4: Elementary operation.

Single coefficients of the resulting matrix are computed by appropriately connecting the processes. For instance, Fig. 5 shows how to compute an entire column of the resulting matrix. By using N of these structures, the N columns of the result are computed (see Fig. 6).

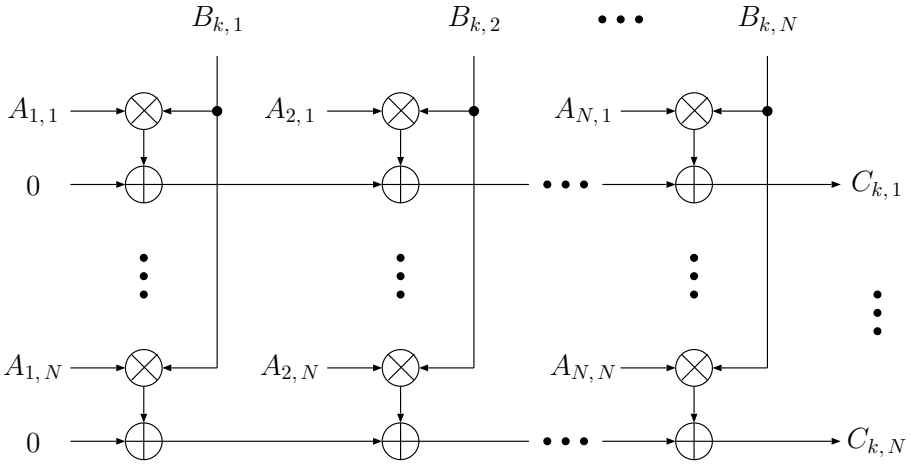

Figure 5: Computation of one column.

To feed the structure and to retrieve the result, an input generator and an output consumer is used. The input generator has ports which provide the factors of the input matrices. Note that each matrix coefficient is the input of N multiplication/addition processes. Since each port can be connected to only one channel, for each matrix coefficient N ports are required. Likewise, there are $N \times N$ ports required to feed zeros to the first layer of multiplication/addition processes. The output consumer has $N \times N$ ports which are connected to the last layer of multiplication/addition processes to receive the result.

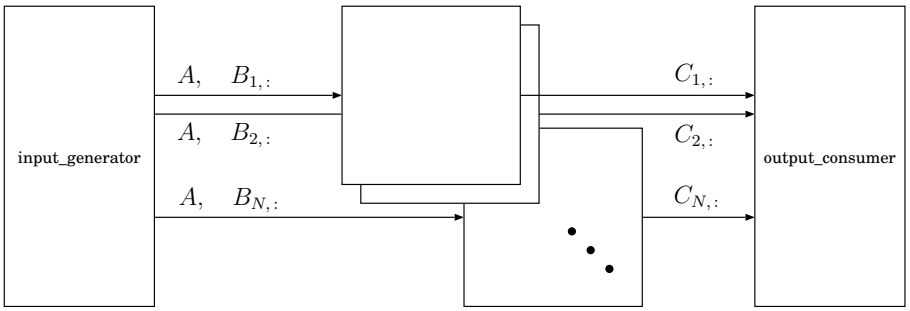

Figure 6: Computation of matrix.

Listings 1 – 3 show the fire() methods of the involved processes. The input generator merely assigns the matrix coefficients to its output ports. The output consumer, in turn, reads the matrix coefficients from its input ports and displays them. The matrix multiplication itself is split up into single multiplications and additions. The dataflow between this elementary operations is described in the corresponding process network. In this example, the functionality of the application is therefore mainly defined by the process network, and not by the C source code.

```
01 int producer_fire(DOLProcess *p)
02 {
03   int row, col, i;
04   float zero = 0.0;
05
06   CREATEPORTVAR(port);
07
08   for (row = 0; row < NUMBER_OF_ROWS_COLS; row++)
09   {
10     for (col = 0; col < NUMBER_OF_ROWS_COLS; col++)
11     {
12       CREATEPORT(port, PORT_ZEROINPUT, 1,
13                  row * NUMBER_OF_ROWS_COLS + col, NUMBER_OF_ROWS_COLS
14                  * NUMBER_OF_ROWS_COLS);
15       printf("%15s: Write to zeroinput_%d: %f\n", "input_generator",
16              row * NUMBER_OF_ROWS_COLS + col, 0.0);
17       DOL_write((void*)port, &zero, sizeof(float), p);
18
19       for (i = 0; i < NUMBER_OF_ROWS_COLS; i++)
20       {
21         CREATEPORT(port, PORT_MATRIXA, 3,
22                    row, NUMBER_OF_ROWS_COLS,
23                    col, NUMBER_OF_ROWS_COLS,
24                    i, NUMBER_OF_ROWS_COLS);
25         printf("%15s: Write to matrixA_%d_%d_%d: %f\n",
```

```
26                 "input_generator", i, row, col,
27                 p->local->matrixA[row][col]);
28         DOL_write((void*)port, &(p->local->matrixA[row][col]),
29                   sizeof(float), p);
30         CREATEPORT(port, PORT_MATRIXB, 3,
31                    row, NUMBER_OF_ROWS_COLS,
32                    col, NUMBER_OF_ROWS_COLS,
33                    i, NUMBER_OF_ROWS_COLS);
34         printf("%15s: Write to matrixB_%d_%d_%d: %f\n",
35                 "input_generator", i, row, col,
36                 p->local->matrixB[row][col]);
37         DOL_write((void*)port, &(p->local->matrixB[row][col]),
38                   sizeof(float), p);
39       }
40     }
41   }
42
43   DOL_detach(p);
44   return -1;
45 }
```

Listing 1: fire() of input generator process.

```
01 int addmult_fire(DOLProcess *p)
02 {
03   float factor1, factor2, summand;
04
05   DOL_read((void*)PORT_FACTOR1, &factor1, sizeof(float), p);
06   DOL_read((void*)PORT_FACTOR2, &factor2, sizeof(float), p);
07   DOL_read((void*)PORT_SUMMAND, &summand, sizeof(float), p);
08   p->local->sum = factor1 * factor2 + summand;
09   DOL_write((void*)PORT_SUM, &(p->local->sum), sizeof(float), p);
10
11   printf("%15s: %f * %f + %f = %f\n",
12          p->local->id, factor1, factor2, summand, p->local->sum);
13
14   return 0;
15 }
```

Listing 2: fire() of multiplication/addition process.

```
01 int consumer_fire(DOLProcess *p)
02 {
03   float matrixC_value;
04   int row, col;
05   CREATEPORTVAR(port);
06
07   for (row = 0; row < NUMBER_OF_ROWS_COLS; row++)
08   {
09     for (col = 0; col < NUMBER_OF_ROWS_COLS; col++)
10     {
11       CREATEPORT(port, PORT_MATRIXC, 2,
12                  row, NUMBER_OF_ROWS_COLS,
```

```
13                  col, NUMBER_OF_ROWS_COLS);
14
15          DOL_read((void*)port, &matrixC_value, sizeof(float), p);
16          printf("%15s: matrixC[%d][%d]: %f\n",
17                  "output_consumer", row, col, matrixC_value);
18        }
19      }
20      return 0;
21  }
```

Listing 3: fire() of output consumer process.

5 Example 5

In example 5, the N-point fast Fourier transform (FFT) is considered. The N-point discrete Fourier transform (DFT) is a block operation which takes N complex-valued (time-domain) input coefficients and transforms them into N complex-valued (frequency-domain) output coefficients. The N-point DFT is defined by:

$$\mathbf{X}[k] = \sum_{n=0}^{N-1} \mathbf{x}[n] \cdot e^{j \cdot \frac{2\pi k n}{N}}, \quad 0 \leq k < N.$$

An efficient implementation of the DFT is the decimation-in-time radix-2 FFT. In this implementation, the computation is split up into $N/2 \cdot \log_2(N)$ 2-point FFTs. The 2-point FFT is shown in Fig. 7.

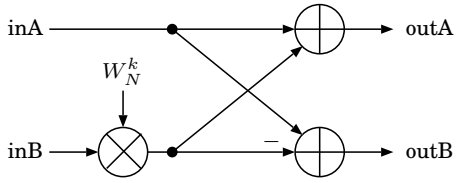

Figure 7: 2-point FFT. W_N^k stands for $e^{j \cdot \frac{2\pi k}{N}}$.

Using the 2-point FFT, higher-order FFTs can be computed by arranging $N/2 \cdot \log_2(N)$ 2-point FFTs and connecting them using a butterfly network (see Fig. 8). The complexity of the FFT is $O(N \log N)$ compared to $O(N^2)$ of the DFT.

To model the FFT, we proceed as follows:

- The input coefficients are ordered in bit-reversed order, that is, the index of the coefficient at channel k can be computed by writing down

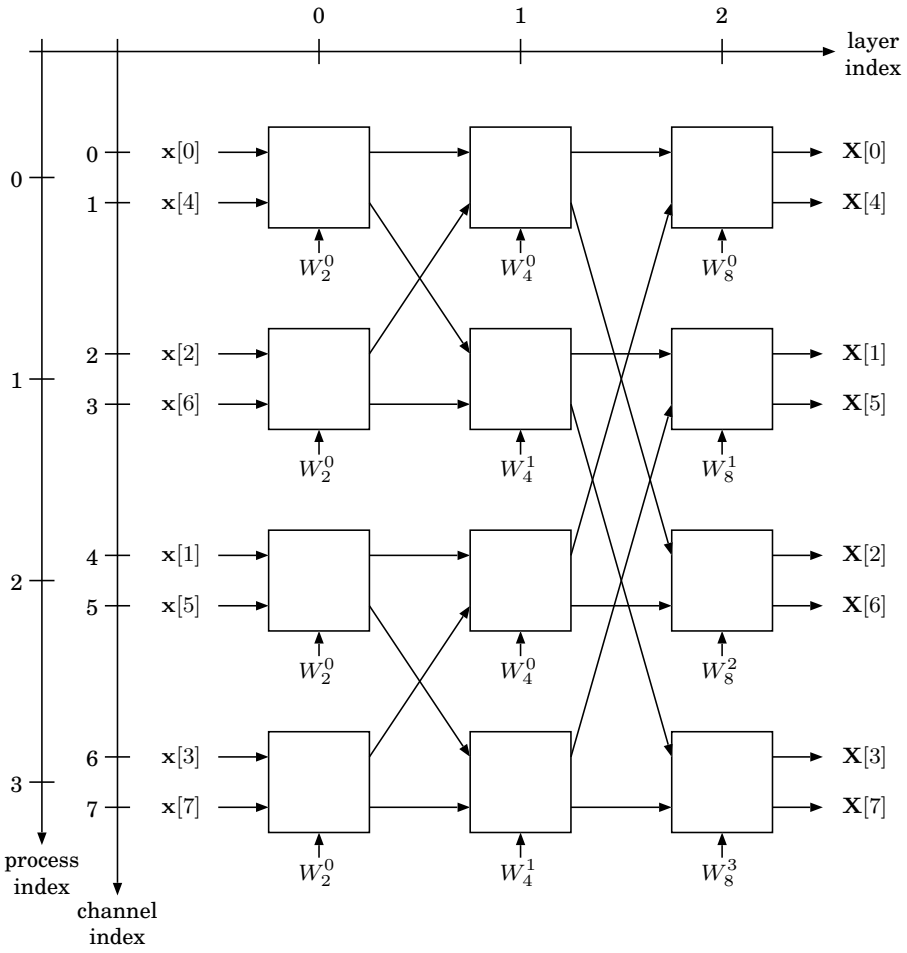

Figure 8: 8-point FFT.

the binary representation of k using $\log_2(N)$ bits and reversing the order of bits. For instance: $3 \rightarrow 011 \rightarrow 110 \rightarrow 6$. This function is defined in a <function> element in the process network.

- The butterfly network is generated by properly connecting the ports of the 2-point FFTs. The index of the channel connected to port inA of a 2-point FFT with process index p and layer index l is given by:

$$\text{index}_{\text{inA}} = 2\left(p \bmod 2^{l-1}\right) + \left\lfloor \frac{p}{2^{l-1}} \right\rfloor + \left\lfloor \frac{p}{2^l} \right\rfloor \cdot \left(2^{l+1} - 2\right).$$

The index of the channel connected to port inB is given by $\text{index}_{\text{inB}} =$

$\text{index}_{\text{inA}} + 2^l$. This function is defined in a <function> element in the process network.

- Each 2-point FFT in the last layer of the network yields two output coefficients. In particular, the indices of the output coefficients are as follows, where p stands for the process index: outA = $\mathbf{X}[p]$ and outB = $\mathbf{X}[p+N/2]$. This function is directly defined in a corresponding <append> element.

- The coefficient W_N^k for a process with index p and located in layer l is $W_{2^{(l+1)}}^{p \bmod 2^l}$. This function is computed in the init() function of the 2-point FFT process.

6 Example 6

In this example, the first order IIR filter is considered. A first order IIR filter can be described by its state equation, as follows, where $\mathbf{x}$ denotes the sequence of input samples and $\mathbf{y}$ denotes the sequence of output samples. Fig. 9 shows a graphical representation of this equation.

$$\mathbf{y}[n] = \mathbf{x}[n] + c \cdot \mathbf{y}[n-1].$$

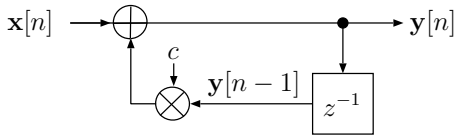

Figure 9: First order IIR filter.

The corresponding process network is shown in Fig. 10. The output of the filter is replicated within the filter process and fed to the consumer on the one hand, and to the filter itself on the other hand.

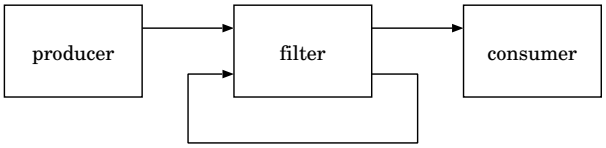

Figure 10: Process network of first order IIR filter.

The order of events in the application is determined by the reads and

writes of the filter process. The code of the init() and fire() method of the filter process is shown below:

```
01 void filter_init(DOLProcess *p)
02 {
03   p->local->zero = 0.0;
04   p->local->factor = 0.5;
05   DOL_write((void*)PORT_OUTB, &(p->local->zero), sizeof(float), p);
06 }
07
08 int filter_fire(DOLProcess *p)
09 {
10   DOL_read((void*)PORT_INA, &(p->local->inA), sizeof(float), p);
11   DOL_read((void*)PORT_INB, &(p->local->inB), sizeof(float), p);
12   p->local->out = p->local->inA + p->local->factor * p->local->inB;
13   DOL_write((void*)PORT_OUTA, &(p->local->out), sizeof(float), p);
14   DOL_write((void*)PORT_OUTB, &(p->local->out), sizeof(float), p);
15
16   return 0;
17 }
```

Listing 4: init() and fire() methods of filter.

7 Example 7

Example 7 extends example 6 to an N-th order IIR filter, as shown in Fig. 11.

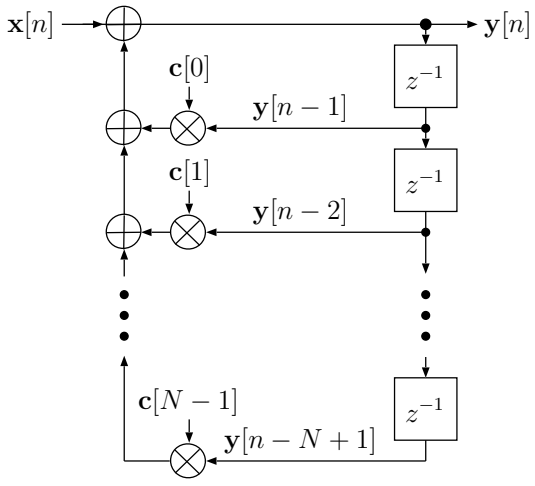

Figure 11: N-th order IIR filter.

The computation can be split into first order filter computations. In particular, a filter process with two inputs (delayed y from previous stage, sum from next stage) and two outputs (delayed y to next stage, sum to

previous stage) is used. That way, the computation can be organized in a process network as shown in Fig. 12. Note that there are $(N + 1)$ filter processes, because the process called filter_0 merely adds the $\mathbf{x}[n]$, and does not contribute to filtering.

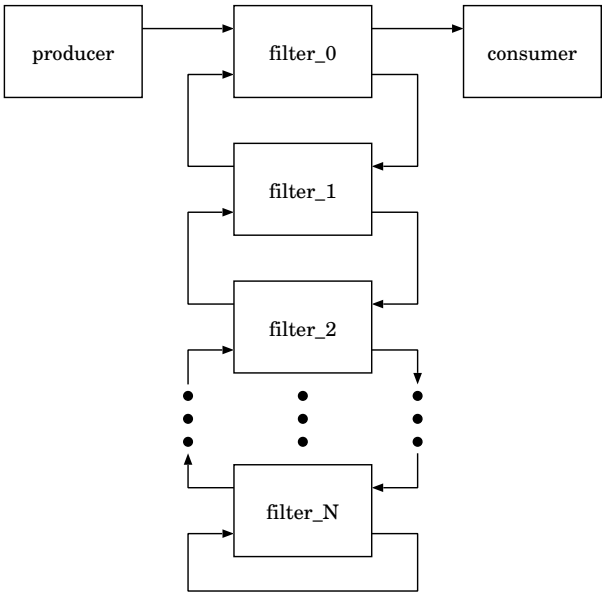

Figure 12: Process network of N-th order IIR filter.

8 Example 8

Example 8 shows how to use the DOL_rtest() and DOL_wtest() primitives, which is shown in Fig. 13.

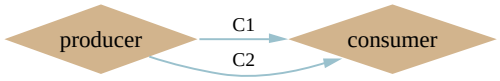

Figure 13: Example 8.

DOL_rtest() and DOL_wtest() are non-blocking test primitives, which can be used to test the status of a FIFO channel, e.g. full or empty. For an output port which connects to the input of a FIFO, only DOL_wtest() is defined. Conversely, DOL_rtest() is only defined for input ports.

In Example 8, the *producer* process has two output ports and will write a sequence of letters to these two ports character by character. It will continue writing to $PORT\_OUTA$ until the $C1$ channel is full, then it will write to $PORT\_OUTB$, i.e, write to channel $C2$. The DOL_wtest() is called in each fire to check whether the $C1$ channel is full, see Line 03 of Listing 5. The *consumer* process will fetch data from both FIFOs only when the $C2$ channel is readable.

Example 8 is an example that is only used to test the validation of these two primitives. The simulation won't stop after the beginning. The reason is the DOL_rtest() in the *consumer* process. Each time, the *consumer* process can fetch data from both FIFOs only when the $C2$ FIFO has data, see Line 24. Then the *consumer* process cannot access the $C1$ channel in which there is data. Therefore, the read notification signal of the $C1$ channel cannot be removed and the SystemC scheduler has to call the *consumer* process again and again.

```
01 int producer_fire(DOLProcess *p) {
02   if (p->local->index < p->local->len) {
03     if (DOL_wtest((void*)PORT_OUTA, 1, p)) {
04       DOL_write((void*)PORT_OUTA, &(p->local->str[p->local->index]),
05                 1, p);
06       printf("p write to port A %c\n",
07              p->local->str[p->local->index]);
08     } else {
09       DOL_write((void*)PORT_OUTB, &(p->local->str[p->local->index]),
10                 1, p);
11       printf("p write to port B %c\n",
12              p->local->str[p->local->index]);
13       }
14     p->local->index++;
15     return 0;
16   } else {
17     DOL_detach(p);
18     return -1;
19   }
20 }
21
22 int consumer_fire(DOLProcess *p) {
23   char c;
24   if (DOL_rtest((void*)PORT_INB, 1, p)) {
25     DOL_read((void*)PORT_INA, &c, sizeof(char), p);
26     printf("from port B: %c\n", c);
27     DOL_read((void*)PORT_INA, &c, sizeof(char), p);
28     printf("from port A: %c\n", c);
29   }
30   return 0;
31 }
```

Listing 5: Source of Example 8.