Exercise 4: SystemC and Virtual Prototyping

Exercise on custom `sc_interface`

Matthias Jung
WS 2017/2018

The source code to start this execise is available here:
`https://github.com/TUK-SCVP/SCVP.Exercise4`

Task 1

Custom Interfaces, Ports and Channels

Figure 1 shows an example for a very simple *Petri Net* (PN). The goal of this exercise is to implement the semantics of PNs in SystemC by using custom interfaces, templated ports and channels. We will reach this goal by going step by step trough this exercise. The *transitions* will be implemented as `sc_modules` and the *places* will be implemented as a custom channels which are connected to ports of the transitions.

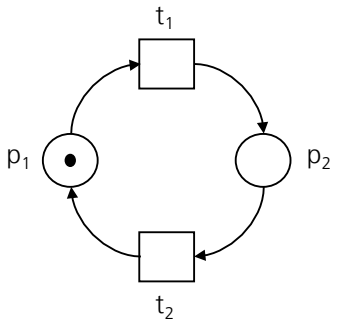

Fig. 1: Simple Petri Net

First, an interface `placeInterface` has to be implemented, which describes the access methods of a place channel by using pure virtual methods. The abstract class `placeInterface` inherits from `sc_inteface` and should have the following pure virtual methods:

```
void addTokens(unsigned int n)
```

```
void removeTokens(unsigned int n)
unsigned int testTokens()
```

Second, a `place` channel should be created, which inherits from `placeInterface`. The channel should have a member variable `unsigned int tokens`, which specifies the current number of tokens on the place. The initial number of tokens should be set in the constructor of the channel `place`.

Next the virtual methods must be specified:

- The method `addTokens(unsigned int n)` should add `n` tokens to the member variable `tokens`.
- The method `removeTokens(unsigned int n)` should subtract `n` tokens from the member variable `tokens`.
- The method `testTokens()` should return the value of `tokens`.

Third, an `SC_MODULE` class `transition` should be defined, which has two `sc_ports` from template type `placeInterface` called `in` and `out`. Furthermore, the `transition` module should have a method `fire()` [1]. This method should check by using the `testTokens` method of the `in` port if it is enabled, i.e. there exists one token in the place. If this is the case, it should printout the following:

```
std::cout << this->name() << ": Fired" << std::endl;
```

And it should remove one token from the `in` port and add a token to the `out` port. If the number of tokens in the place is 0, the following should be printed:

```
std::cout << this->name() << ": NOT Fired" << std::endl;
```

In order to test our initial Petri Net implementation, we build the PN shown in Figure 1 with the following testbench code:

```
SC_MODULE(toplevel)
{
    public:
    transition t1, t2;
    place p1, p2;
```

[1] For the beginning of this exercise, we assume that all weights of the arcs are one and that the petri net has no forks or joins, a place is connected to exatly one transitions output and one transitions input, e.g. Figure 1.

```
    SC_CTOR(toplevel) : t1("t1"), t2("t2"), p1(1), p2(0)
    {
        SC_THREAD(process);

        t1.in.bind(p1);
        t1.out.bind(p2);
        t2.in.bind(p2);
        t2.out.bind(p1);
    }

    void process()
    {
        while(true)
        {
            wait(10,SC_NS);
            t1.fire();
            wait(10,SC_NS);
            t1.fire();
            wait(10,SC_NS);
            t2.fire();
            sc_stop();
        }
    }
};
```

Observe how the PN is constructed by the SC_CTOR of the `toplevel` module. Create one instance of `toplevel` with the name `t` in the `sc_main` function. After the compilation of your code and running the you should see the following output:

```
t.t1: Fired
t.t1: NOT Fired
t.t2: Fired
```

After firing `t1` it cannot be fired again, because a token is missend on the `t1`'s input place.

Multiports

In order to have more than one port for `in` and `out` we template the `transition` module in the following way, in order to support multiple input ports:

```
template<unsigned int N=1, unsigned int M=1>
SC_MODULE(transition)
{
    public:
    sc_port<placeInterface, N, SC_ALL_BOUND> in;
    sc_port<placeInterface, M, SC_ALL_BOUND> out;

    ...
}
```

The template parameter N denotes the number of input ports and M denotes the number of output ports respectively.

The method `fire()` must be modified to the previos example. It should check if there is one token on each input port, using the `in[i]->testTokens()` method call (where $0 \leq$ `i` $<$ N).

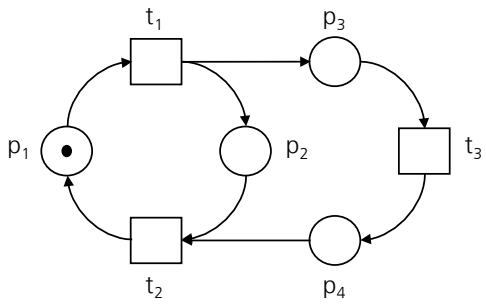

Fig. 2: Simple Petri Net

If there are enough tokens, it should remove one token from each input port and add a one token to each output port. Also the printing of the firing should be done as in the previous example. In order to test our implementation, we implement the PN shown in Figure 2 like in the following testbench:

```
SC_MODULE(toplevel)
{
    public:
    transition<1,2> t1;
    transition<2,1> t2;
    transition<1,1> t3;
    place p1, p2, p3, p4;

    SC_CTOR(toplevel) : t1("t1"), t2("t2"), t3("t3"),
                        p1(1), p2(0), p3(0), p4(0)
    {
        SC_THREAD(process);

        t1.in.bind(p1);
        t1.out.bind(p2); // 0
        t1.out.bind(p3); // 1
        t2.in.bind(p2);  // 0
        t2.in.bind(p4);  // 1
        t2.out.bind(p1);
        t3.in.bind(p3);
        t3.out.bind(p4);
    }

    void process() {
        while(true) {
            wait(10,SC_NS);
            t1.fire();
            wait(10,SC_NS);
            t2.fire();
            wait(10,SC_NS);
            t3.fire();
            wait(10,SC_NS);
            t2.fire();
            sc_stop();
        }
    }
};
```

Note that the order of binding determines to which port number of the port array the channels are bound.

The output should be the following:

```
t.t1: Fired
t.t2: NOT Fired
t.t3: Fired
t.t2: Fired
```

Note that `t2` cannot fire until `t3` is fired.

Templated Channels

As the next step, we want to include weights to the channels at the input arc of a place and to the output arc of a place. First, we change the interface class that the methods are looking like this:

```
void addTokens();
void removeTokens();
bool testTokens();
```

As before for the `toplevel` module, we use templates for the `place` channel:

```
template<unsigned int I=1, unsigned int O=1>
class place : public placeInterface
...
```

Where, `I` denotes the input weight of a place and `O` the output weight of a place. The methods of the `place` channel have to be changed according to the new `placeInterface`:

- The method `addTokens()` should now add `I` tokens to `tokens`.
- The method `removeTokens()` should now subtract `O` tokens from `tokens`
- The method `testTokens()` should test if the number of `tokens` is greater or equal `O`.

The `transition` module should be adapted accordingly. To test use the following code:

```
SC_MODULE(toplevel)
{
    public:
    transition<1,2> t1;
    transition<2,1> t2;
    transition<1,1> t3;
    place<1,1> p1, p2, p3, p4;

    SC_CTOR(toplevel) : t1("t1"), t2("t2"), t3("t3"),
                        p1(1), p2(0), p3(0), p4(0)
    {
        SC_THREAD(process);

        t1.in.bind(p1);
        t1.out.bind(p2); // 0
        t1.out.bind(p3); // 1
        t2.in.bind(p2);  // 0
        t2.in.bind(p4);  // 1
        t2.out.bind(p1);
        t3.in.bind(p3);
        t3.out.bind(p4);
    }

    void process() {
        while(true) {
            wait(10,SC_NS);
            t1.fire();
            wait(10,SC_NS);
            t2.fire();
            wait(10,SC_NS);
            t3.fire();
            wait(10,SC_NS);
            t2.fire();
            sc_stop();
        }
    }
};
```

Modelling a Single Memory Bank

With the current code base model the single memory bank example from the lecture, as shown in Figure 3. Use the same names for transitions and places as shown in the picture.

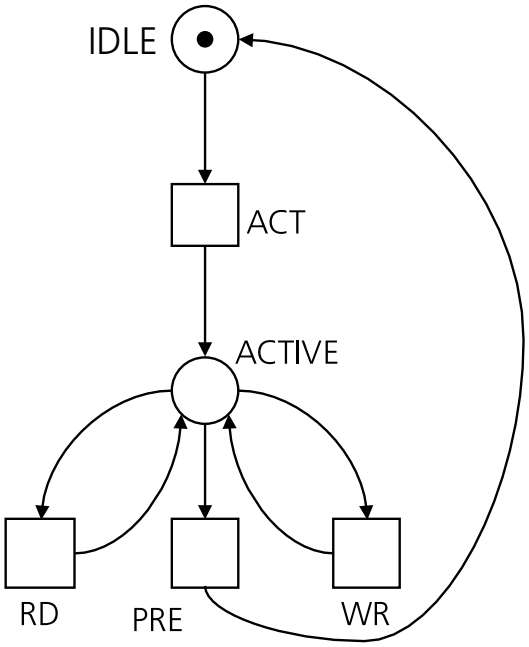

Fig. 3: Single Memory Bank Example

Test your code with the following process:

```
wait(10,SC_NS);
ACT.fire();
wait(10,SC_NS);
ACT.fire();
wait(10,SC_NS);
RD.fire();
wait(10,SC_NS);
WR.fire();
wait(10,SC_NS);
PRE.fire();
wait(10,SC_NS);
ACT.fire();
sc_stop();
```

Implementation of Inhibitor Arcs

In order to implement inhibitor arcs, add an additional template parameter `L` for the number of inhibitor inputs,

```
template<unsigned int N=1, unsigned int M=1, unsigned int L=0>
```

and an additional `sc_port` to the `transition` module:

```
sc_port<placeInterface, L, SC_ZERO_OR_MORE_BOUND> inhibitors;
```

Additionally to the firing check for enough tokens on the input ports, it is checked for all inhibitor ports that there are no tokens on the connected channels by using the `testTokens()` method. In other words, the firing is only performed, if there are enough tokens and no tokens on places that would inhibit the firing. Adjust the `fire()` method accordingly. Why we used the `SC_ZERO_OR_MORE_BOUND` flag?

Building Hierarchical PNs

To finish the exercise, create `sc_module` called `subnet`, which implements the PNs in the green boxes of Figure 4 [2].

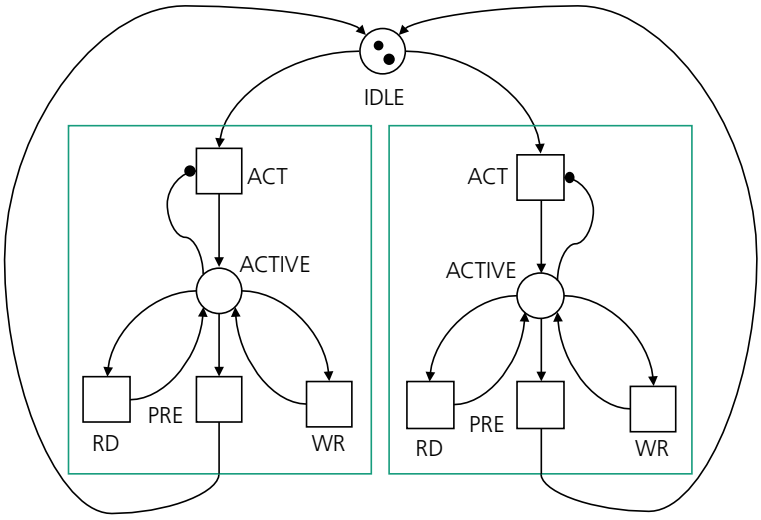

Fig. 4: Two Memory Bank Example with Subnets

[2] Hint: you can also bind input with input ports and output with output ports

After that implement the `toplevel` module in such a way, that it includes two instances of `subnet`, called `s1` and `s2`. Take care that all ports are bound correctly. You can test your subnet again with the following stimuli:

```
wait(10,SC_NS);
s1.ACT.fire();
wait(10,SC_NS);
s1.ACT.fire();
wait(10,SC_NS);
s1.RD.fire();
wait(10,SC_NS);
s1.WR.fire();
wait(10,SC_NS);
s1.PRE.fire();
wait(10,SC_NS);
s1.ACT.fire();
wait(10,SC_NS);
s2.ACT.fire();
wait(10,SC_NS);
s2.ACT.fire();
wait(10,SC_NS);
s1.PRE.fire();
wait(10,SC_NS);
s2.PRE.fire();
wait(10,SC_NS);
sc_stop();
```