Exercise 4: SystemC and Virtual Prototyping

Exercise on Custom `sc_interface`

Lukas Steiner
WS 2023/2024

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 going step by step through the exercise. *Transitions* will be implemented as `SC_MODULE`s and *places* will be implemented as custom channels that are connected to ports of the transitions.

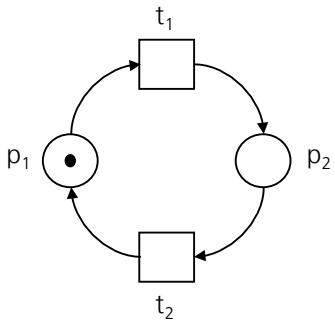

Fig. 1: Simple Petri Net

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

```
virtual void addTokens(unsigned int n) = 0;
virtual void removeTokens(unsigned int n) = 0;
virtual unsigned int testTokens() = 0;
```

Note: a virtual function is specified *pure* by using the *pure specifier* "`= 0`".

Second, a `place` channel should be created inside `place.h`, which inherits from `placeInterface`. The channel should have a member variable `unsigned int tokens`, which specifies the current number of tokens in the place. The initial number of tokens should be passed as a parameter to the constructor and then set in the constructor of the channel `place`.

Next the virtual functions have to be implemented:

- 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 the member variable `tokens`.

Third, an `SC_MODULE` called `transition` should be defined in `transition.h`. As usual, the constructor of the module must be defined, but there is no need to register any member function with the SystemC kernel, i.e., no need to call `SC_METHOD()` or `SC_THREAD()` for this module.

The module has two `sc_ports` from template type `placeInterface` called `in` and `out`. Furthermore, the `transition` module should have a member function `fire()` [1]. This function should check if the transition is enabled by using the method `testTokens` of the `in` port, i.e., if there exists at least one token in the place. In this case it should print

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

remove one token from the `in` port and add a token to the `out` port. If the number of tokens in the place is zero it should print

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

[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, i.e., a place is connected to exactly one transition's output and one transition's input, e.g., Figure 1.

In order to test our initial Petri Net implementation build an SC_MODULE called `toplevel` in `main.cpp` that reflects the PN shown in Figure 1. As a test bench the module should use the following function as an SC_THREAD.

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

Create one instance of `toplevel` with the name `t` in the `sc_main` function. After compiling and running your code you should see the following output:

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

After firing `t1` the transaction cannot be fired again because a token is missing in the 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 function `fire()` must be modified to the previous example. It should check if there is one token in each input port using the `in[i]->testTokens()` method call (where $0 \leq$ `i` $<$ N).

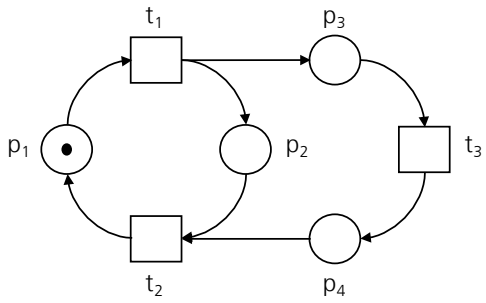

Fig. 2: Simple Petri Net

If there are enough tokens it should remove one token from each input port and add one token to each output port. The printing when firing should be done as in the previous example. In order to test our implementation, implement the PN shown in Figure 2 and use the following function as a test bench (SC_THREAD).

```
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.

After running the application you should see the following output:

```
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 next step we want to add weights to the channels at the input arc of a place and at the output arc of a place. First, we change the methods of the interface class as follows:

```
virtual void addTokens() = 0;
virtual void removeTokens() = 0;
virtual bool testTokens() = 0;
```

As for the `toplevel` module we use templates for the `place` channel

```
template<unsigned int Win = 1, unsigned int Wout = 1>
class place : public placeInterface
...
```

where `Win` denotes the input weight of a place and `Wout` 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 `Win` tokens to the member variable `tokens`.

- The method `removeTokens()` should now subtract `Wout` tokens from the member variable `tokens`.
- The method `testTokens()` should test if the number of `tokens` is greater than or equal to Wout.

The `transition` and `toplevel` modules should be adapted accordingly. By using weights of 1 the new application output should be identical to the previous one.

Modelling a Single Memory Bank

With the current code base we want to model the single memory bank example from the lecture, as shown in Figure 3. Create a `toplevel` module that reflects the memory bank; make sure to use the same names for transitions and places as shown in the picture.

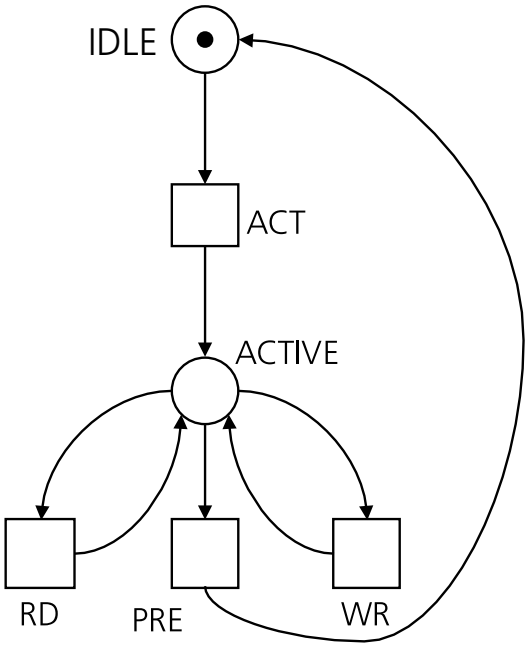

Fig. 3: Single Memory Bank Example

Test your code with the following process:

```
void process()
{
    while (true)
```

```
    {
        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();
    }
}
```

The application output should be the following:

```
t.ACT: Fired
t.ACT: NOT Fired
t.RD: Fired
t.WR: Fired
t.PRE: Fired
t.ACT: Fired
```

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;
```

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

Building Hierarchical PNs

To finish the exercise create an `SC_MODULE` called `subnet` in `subnet.h` that implements the PNs in the green boxes of Figure 4[2].

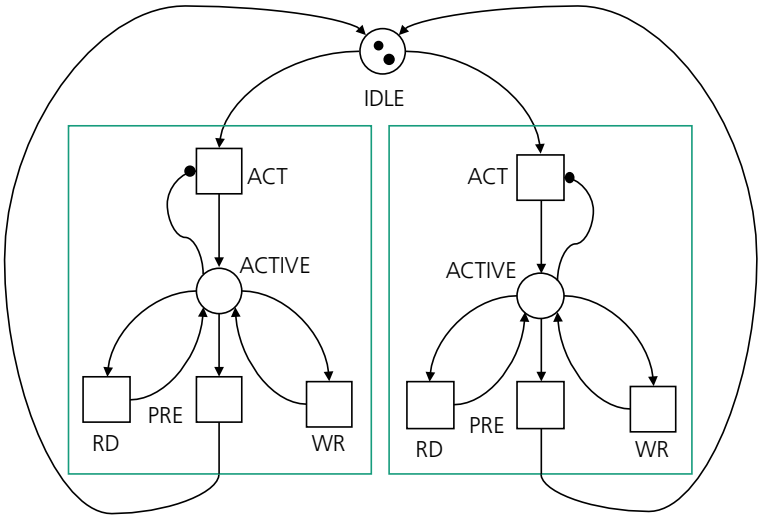

Fig. 4: Two Memory Bank Example with Subnets

Afterwards, 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 implementation with the following process:

```
void process()
{
    while (true)
    {
        wait(10, SC_NS);
        s1.ACT.fire();
```

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

```
        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();
    }
}
```

As an output you should see the following:

```
t.s1.ACT: Fired
t.s1.ACT: NOT Fired
t.s1.RD: Fired
t.s1.WR: Fired
t.s1.PRE: Fired
t.s1.ACT: Fired
t.s2.ACT: Fired
t.s2.ACT: NOT Fired
t.s1.PRE: Fired
t.s2.PRE: Fired
```