RVC CAL Tutorial

RVC CAL is a standardized version of the CAL Actor Language which implements dataflow model of computation.

All the examples of this tutorial are written for and tested with ORCC tools.

Open RVC CAL Compiler (ORCC) is an open-source Integrated Development Environment based on Eclipse and dedicated to dataflow programming. The primary purpose of Orcc is to provide developers with a compiler infrastructure to allow software/hardware code to be generated from dataflow descriptions.

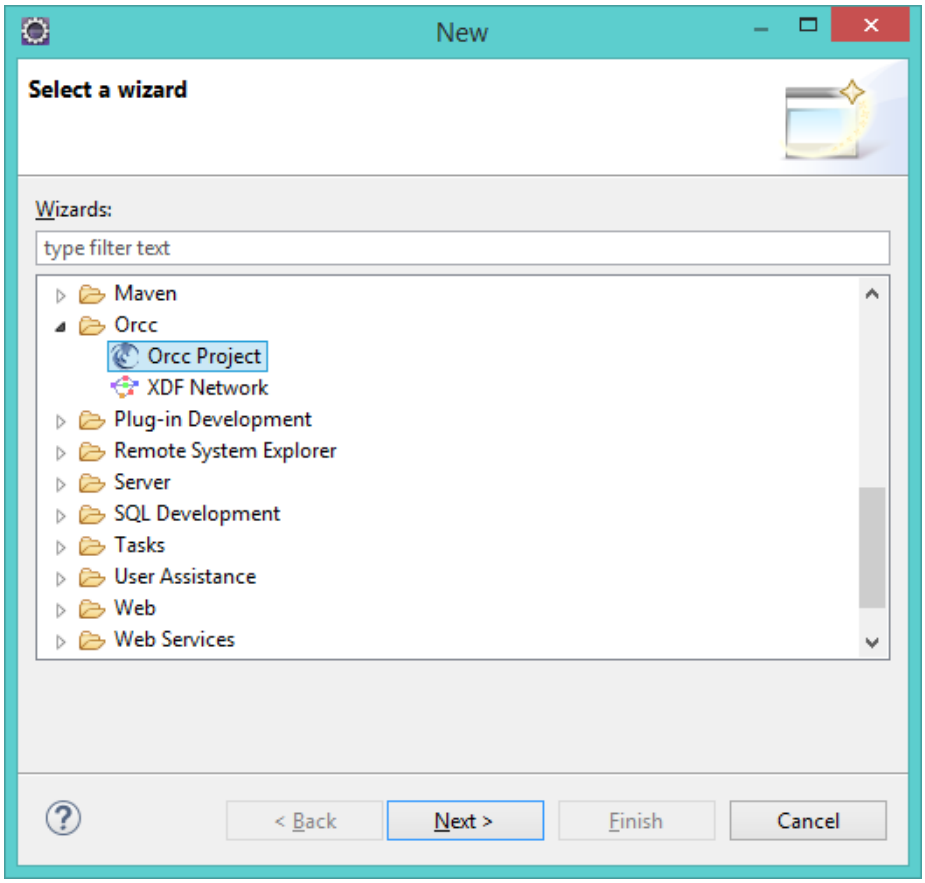
List of contents

Lesson 0. Hello world

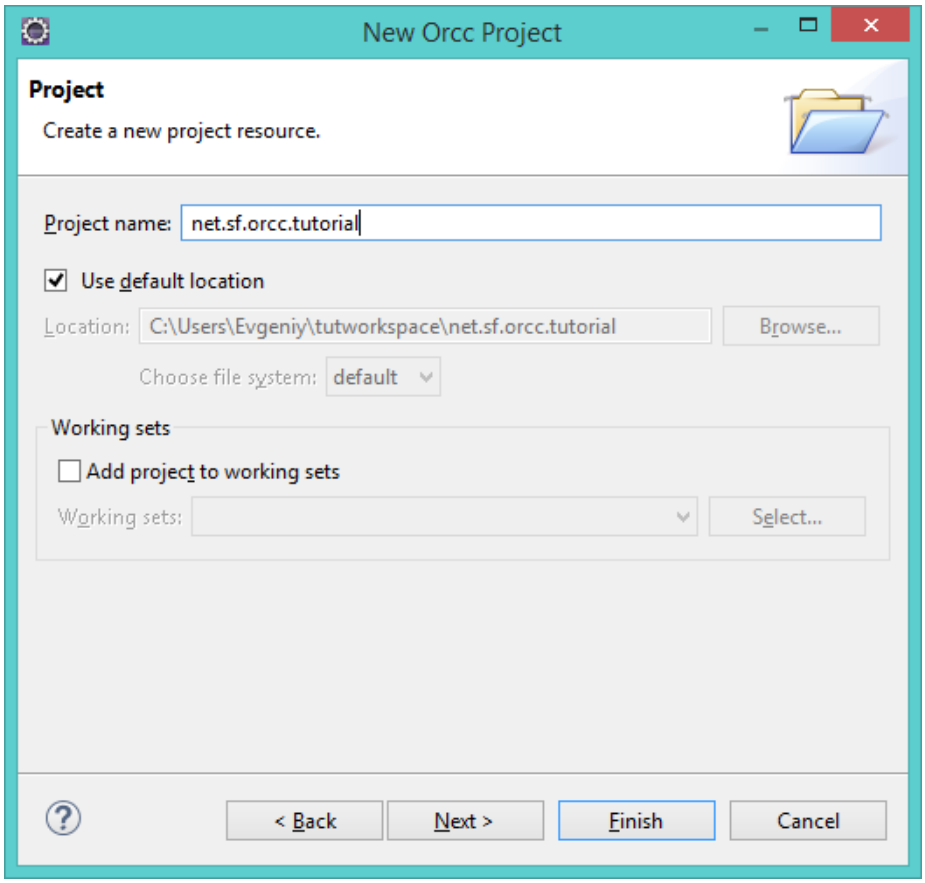
Creating new project

First you need to create new ORCC project.

In the menu “File > New > Other...” chose “ORCC > Orcc Project”



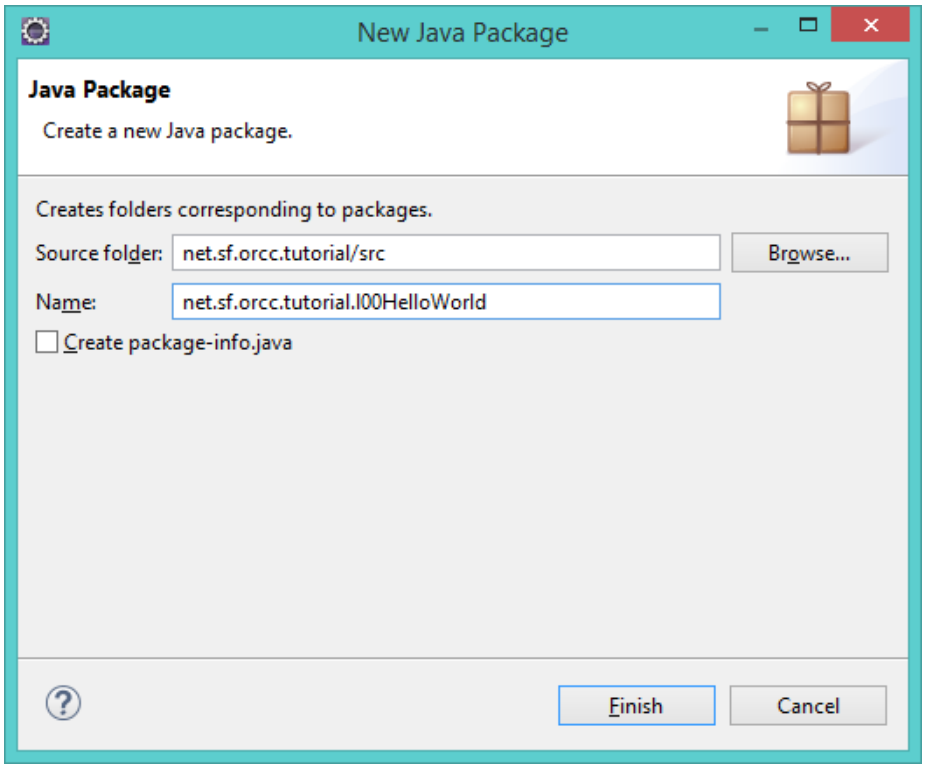
Specify the name of the project and click finish.



You will see that default src directory has been added to the created project.

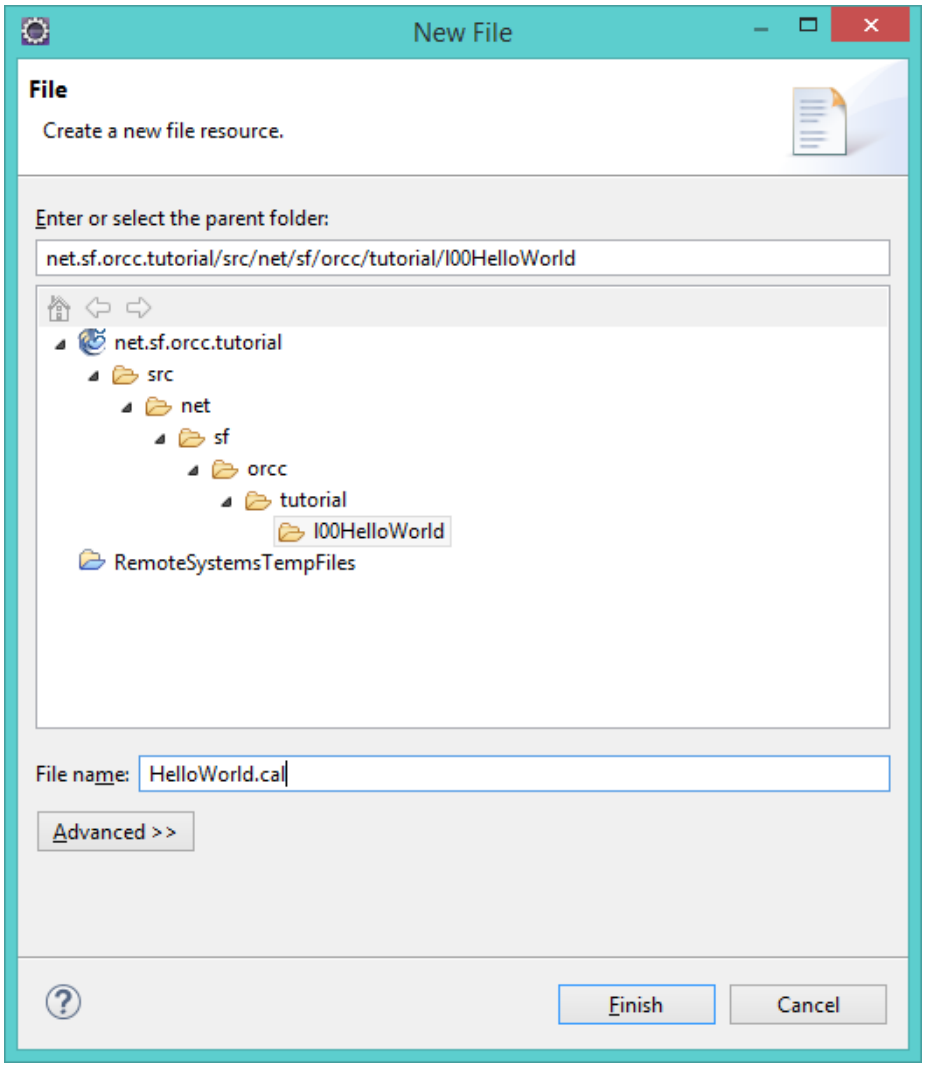
Create new package

Right Click on the src folder in Project Explorer pane. New > Package



Creating Actor

In Project explorer pane, select package you just created. Then click menu "File > New > File..."
Specify the name and the extension .cal



After file was created, add the following code there and save it. Eclipse will automatically compile the file.

```
package net.sf.orcc.tutorial.l00HelloWorld;

actor HelloWorld () int In ==> :
    action ==>
    do
        print("Hello World!\n");
    end
end
```

The code above implements the actor named `HelloWorld()` which takes the input stream of tokens of type `int`.

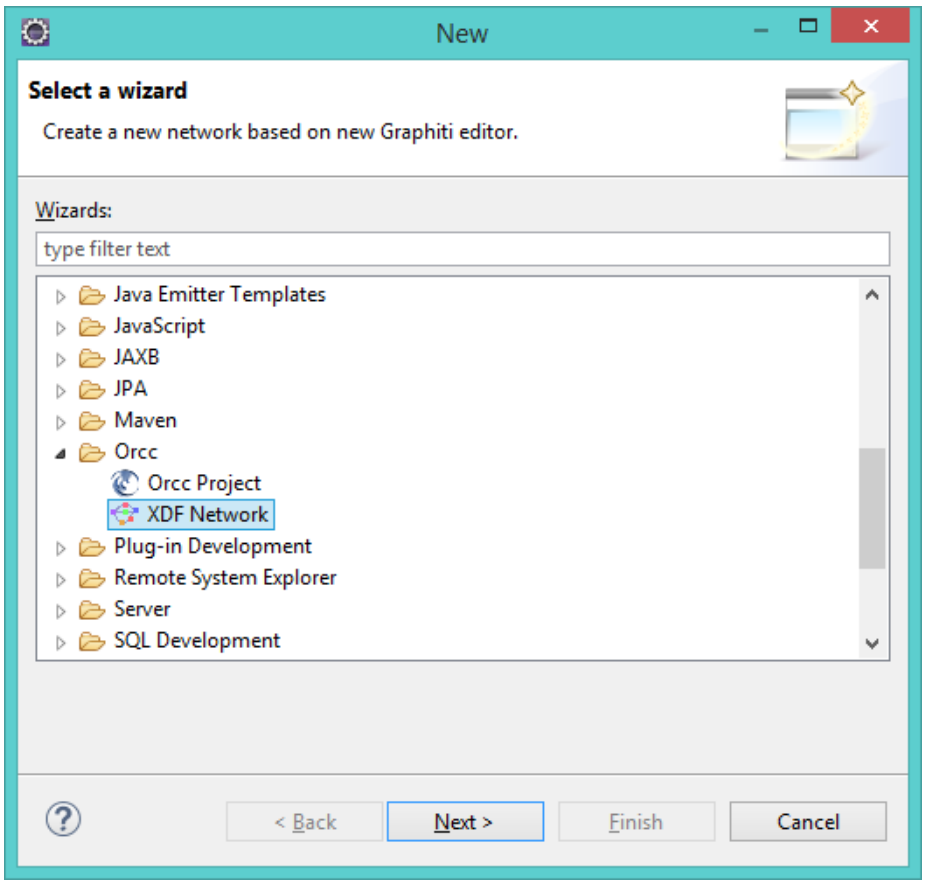
The action within this actor always executes regardless to any input or other conditions and prints the string to the default output.

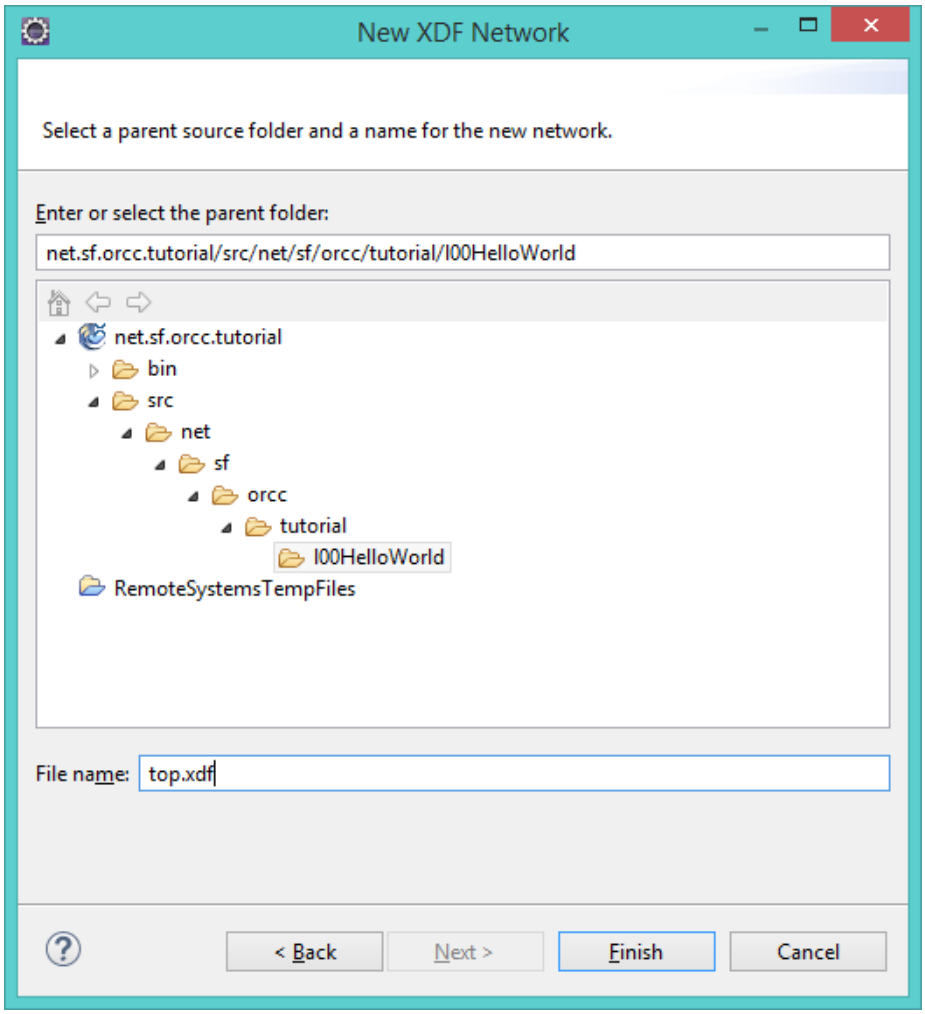
You will find detailed explanation of the syntax in next lessons.

Creating Network

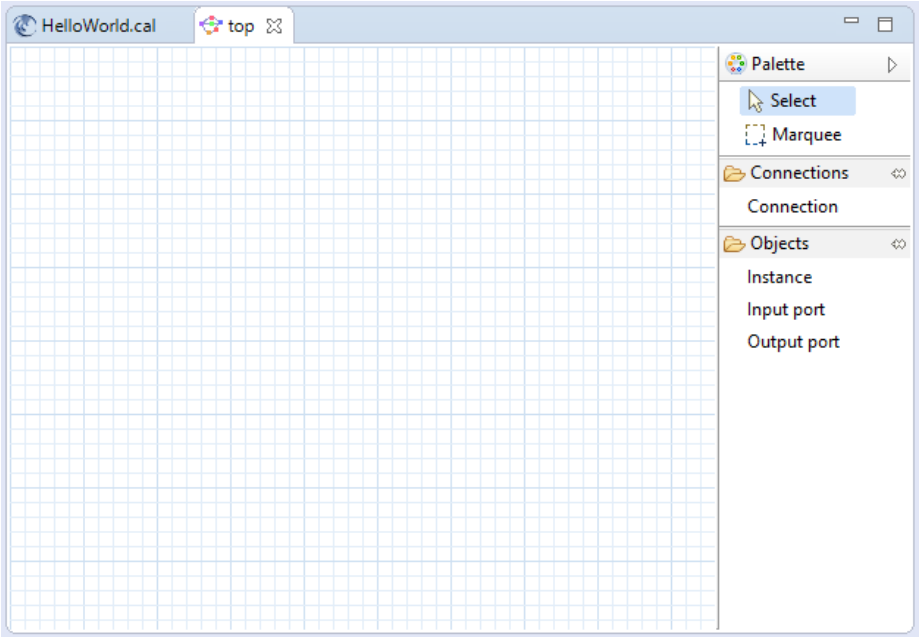
RVC CAL is a language which implements the Dataflow paradigm. This means that in order to run the application you need to build a network of actors. In our case the network will be degenerate and will consist of only one actor.

To build a network you need to create new XDF file. Go to File > New > Other then select Orcc > XDF Network.

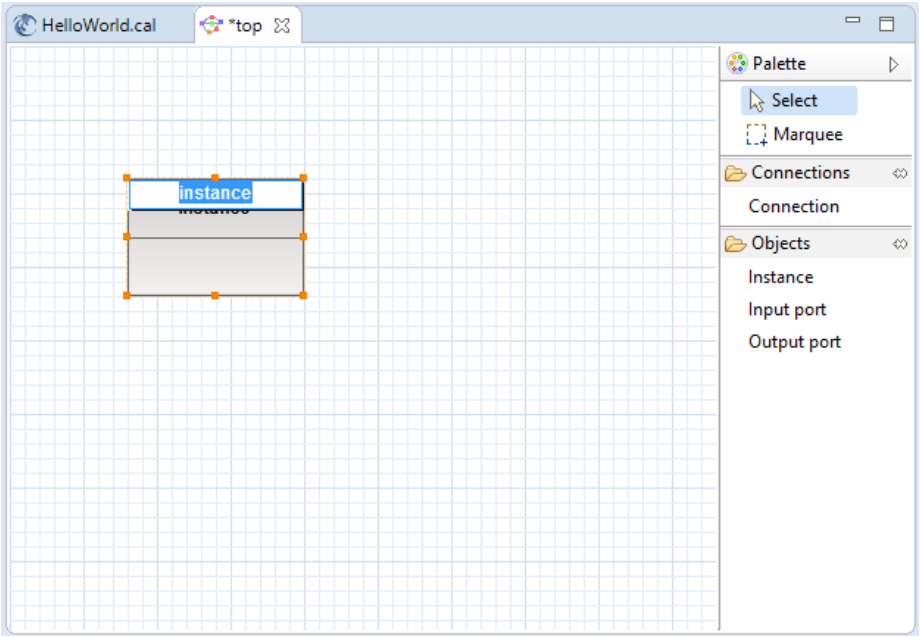




After empty XDF was created you have to add an instance of an actor.



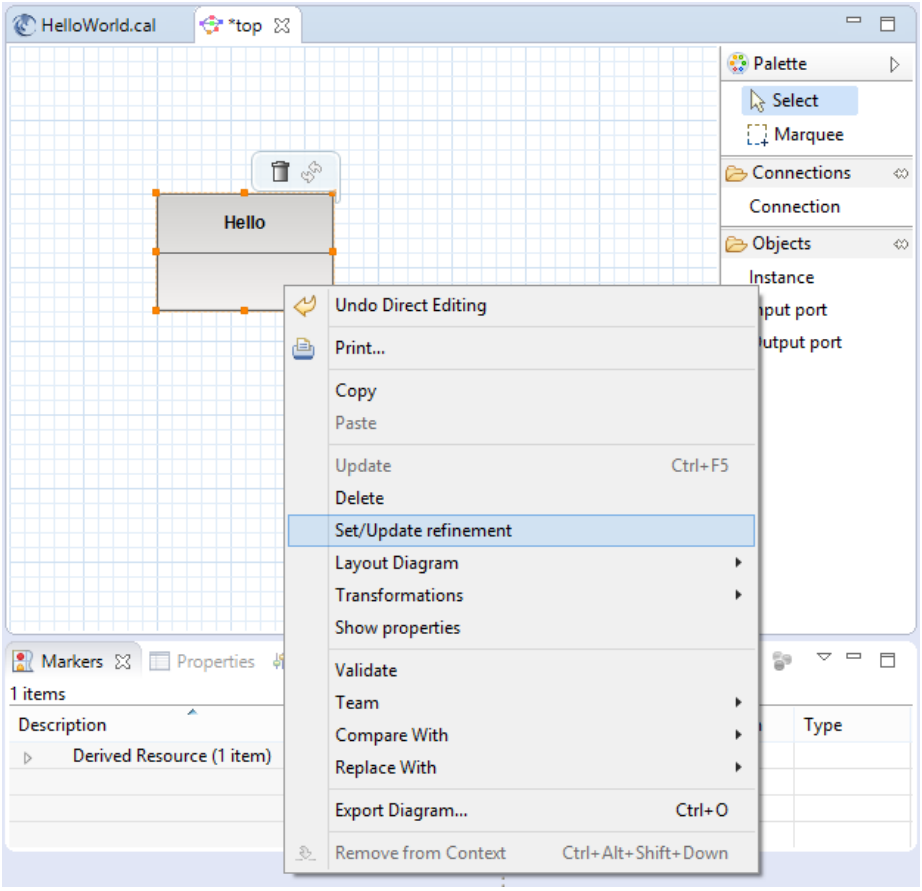
Click on Objects > Instance in the palette, and then click on XDF file area to add an Instance to your network.



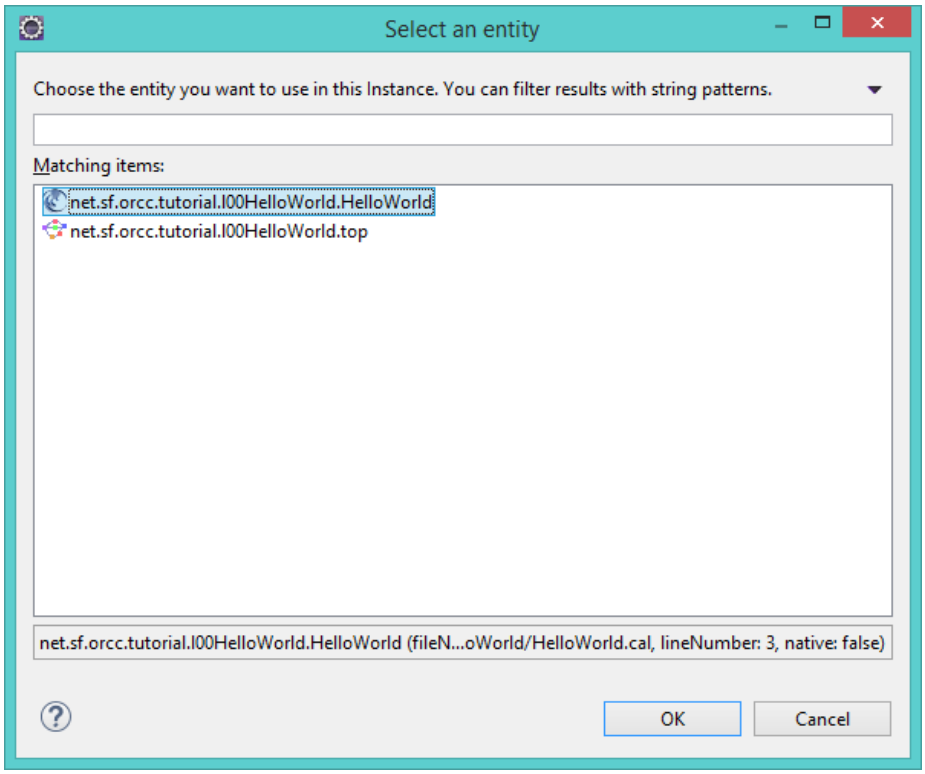
Name your instance “Hello”

Now you have to link this instance to the actor created before. Right-click on

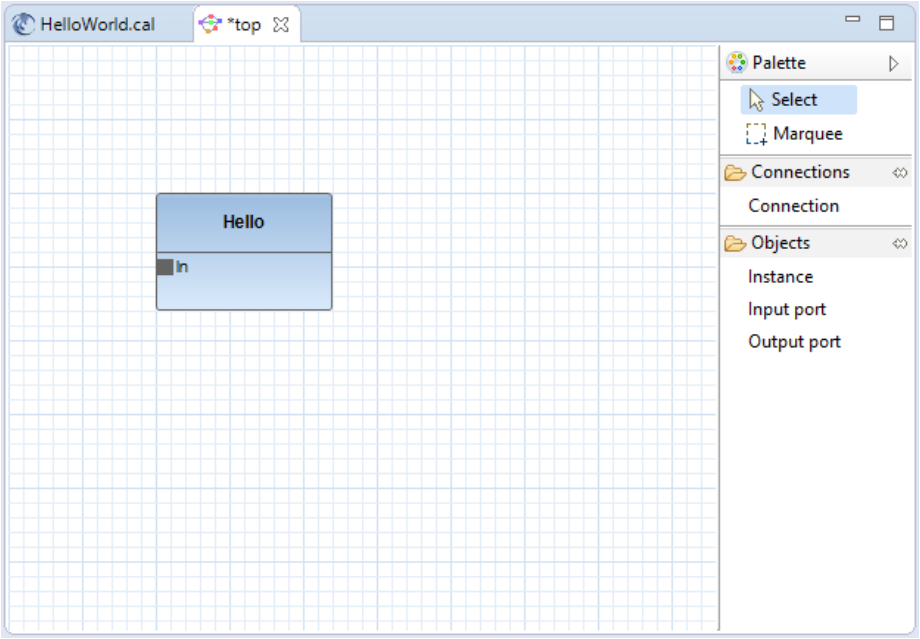
the instance and select Set/Update Refinement.



Then, select the “HelloWorld” actor in the newly opened box.



After your validation, “Hello” should be displayed in blue (meaning that your instance is assigned to an actor).

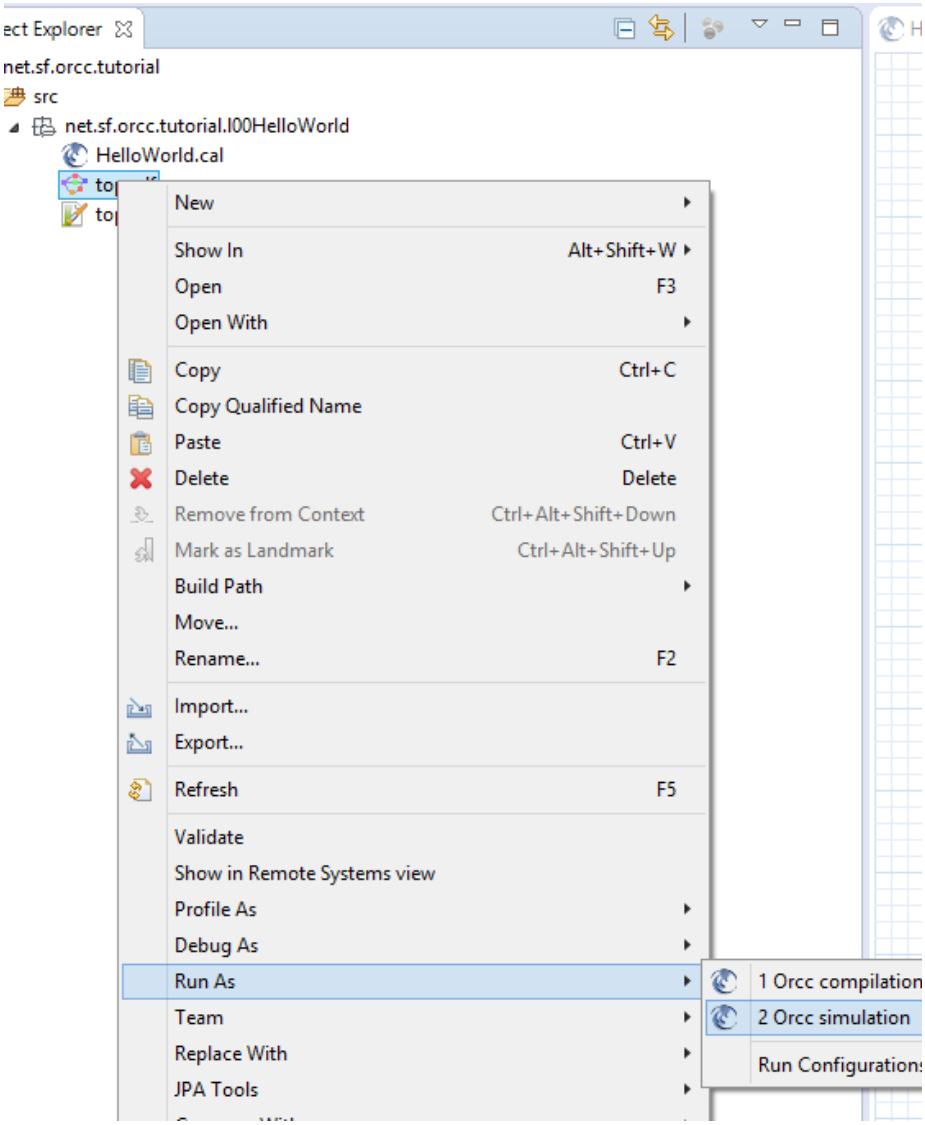


As you can see our instance has an input port, even thought we did not specify it. This is normal.

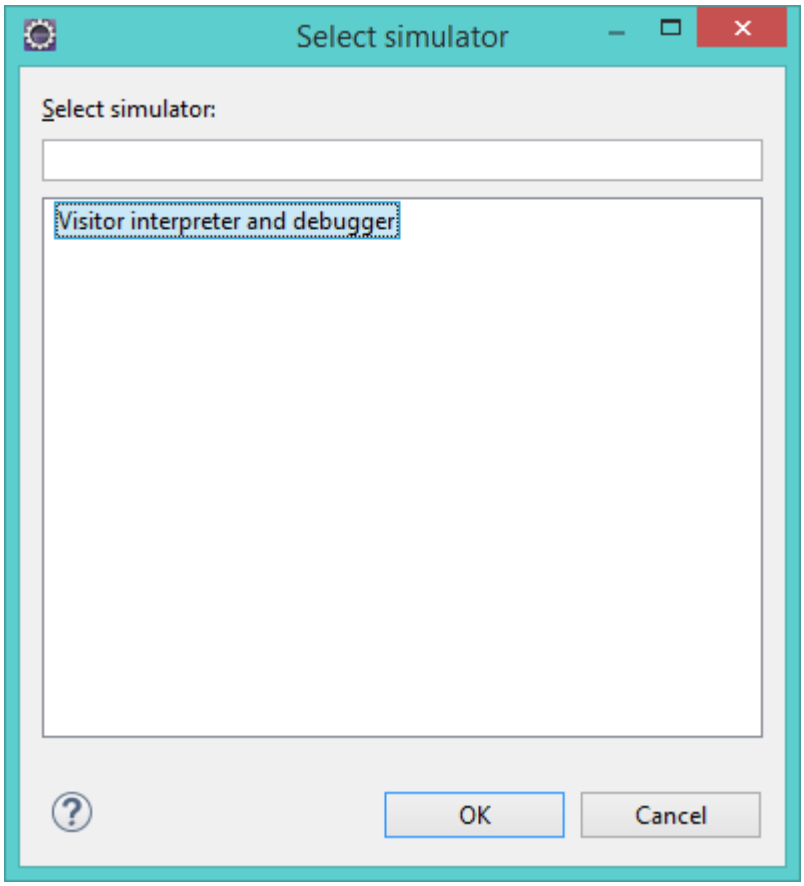
Don't forget to save all the files. Compilation will be done automatically.

Running simulation

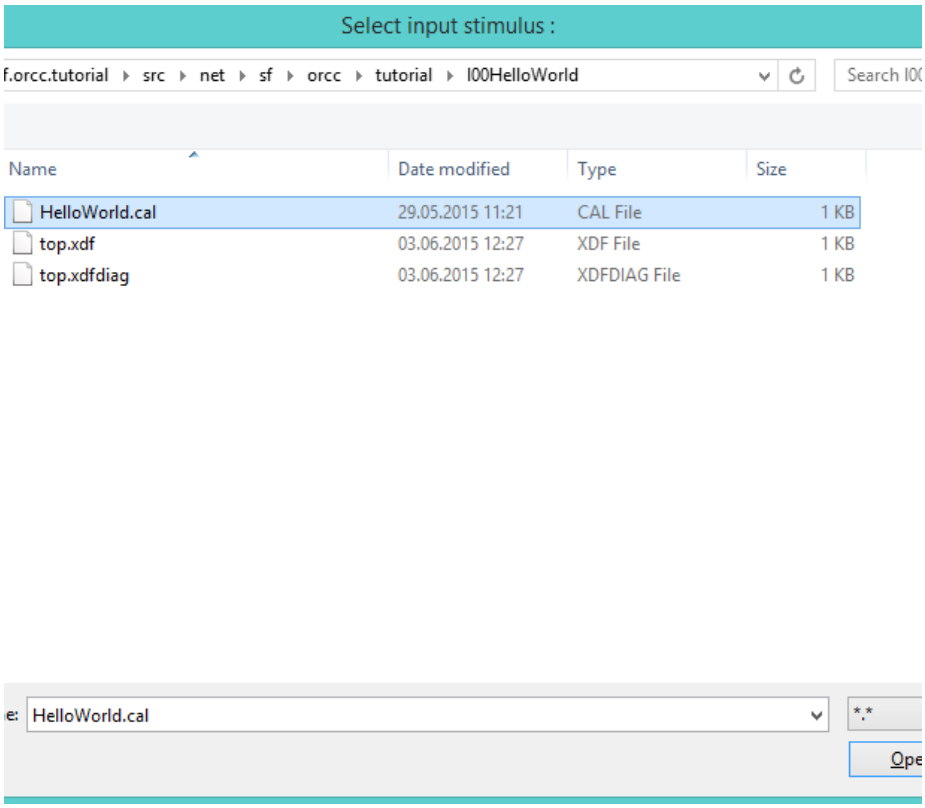
Right-click on the XDF file and (Run As > Orcc simulation).



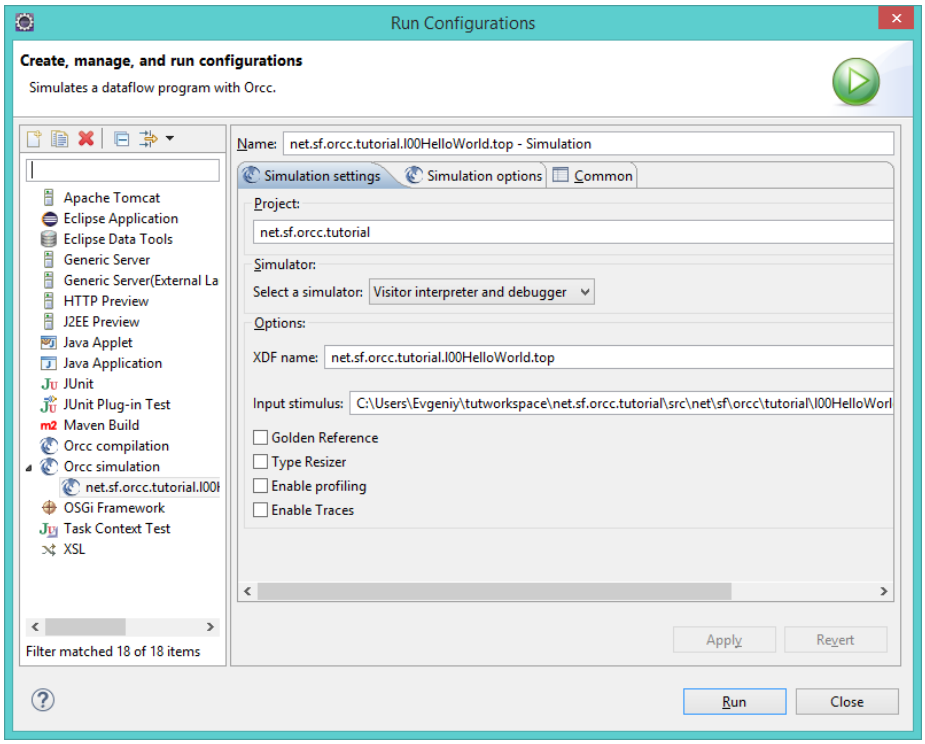
In the "Select simulator" window, click OK.



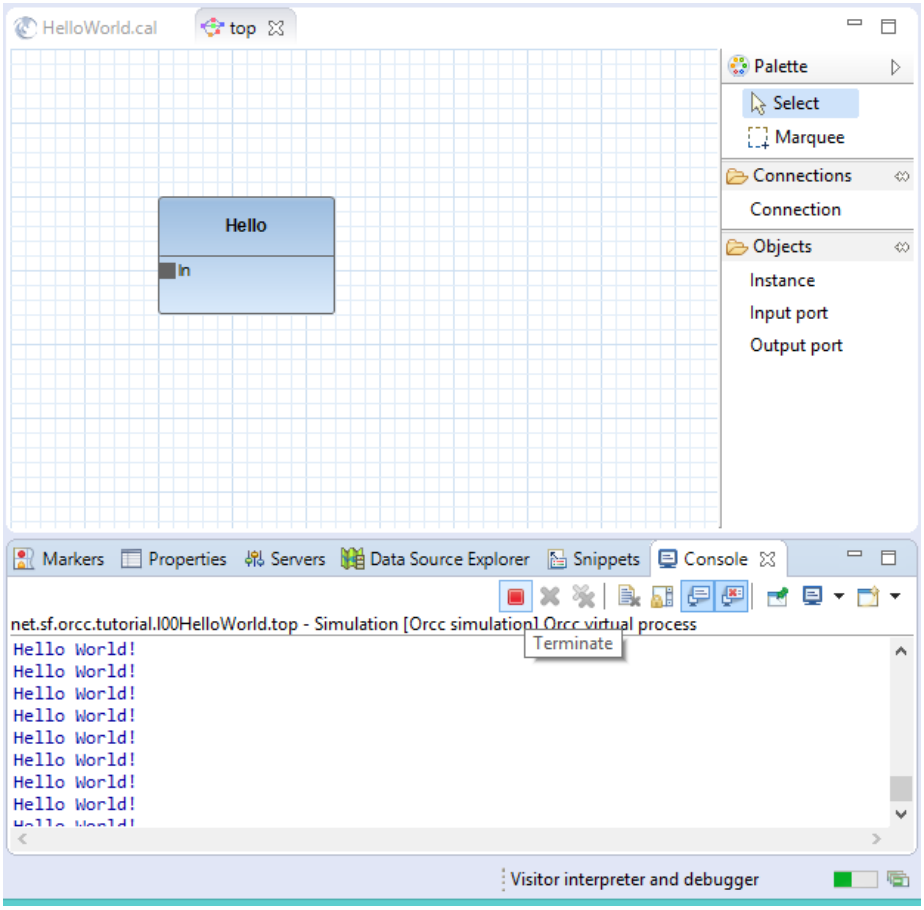
In the “Select input stimulus” window, select a random file (it will not be used by our example).



In the Run configuration wizard just click on Run.



You should see that in Eclipse's console:



Main page | Next lesson

Lesson 1. Simple Actor

Structure of actors *Actors* perform their computation in a sequence of steps called *firings*. In each of those steps

the actor:

1. may consume tokens from its input ports
2. may produce tokens at its output ports

3. may modify its internal state (this is described in further lessons)

Describing an actor involves describing its interface to the outside, the ports, the structure of its internal state, as well as the steps it can perform, what these steps do (in terms of token production and consumption, and the update of the actor state), and how to pick the step that the actor will perform next.

The simplest actor The simplest actor just copies a token from the input to the output unchanged.

```
package net.sf.orcc.tutorial.l01SimpleActor;

actor ID () int In ==> int Out :
  first: action In: [a] ==>  Out: [a] end
end
```

In the first line we specify the package.

The main entity in RVC CAL is an *actor*. In the example above you see that to describe an actor one should use keyword `actor` followed by the name of the actor and parameters in parentheses (empty in this example).
Then you specify input ports before the sign `==>`, and output ports after this sign. RVC CAL is a *statically typed* language, so you need explicitly define type for each variable, i.e. in the line `actor ID () int In ==> int Out :` ports `In` and `Out` both are of type `int`.
The colon at the end of the line marks the start of the actor body which is bounded by the keyword `end` from the other side.

Inside of the body each actor has one or more *actions*, which execute (*fire*) at one step each. Actions may (or may not) *consume* input tokens and *produce* output tokens at each step.
Syntax of describing an action in RVC CAL is the following: `first:` is an *optional* identifier of an action, which can be omitted. After keyword `action` and in front of `==>` sign you see the *input pattern*.

```
actor <ActorIdentifier> () <input ports> ==> <output ports> :
  [<ActionIdentifier>:] action <input pattern> ==> <output expression> end
end
```

Input pattern specifies how many tokens to consume from which ports and how to call these tokens in the rest of the action. The input pattern in `ID` actor is `In: [a]`. It tells the action to consume one token from the input `In` and name it `a` within the action body. *Input pattern* of action realize the idea of pattern matching.

The expression following `==>` sign is an *output expression*. It defines the number and values of output tokens which will be produced on each output port by each *firing* of the action. In this example `Out: [a]` is an *output expression*. It defines that exactly one token will be produced on output port `Out`, and the value of that token will be `a`.

It is important to understand the difference between *input pattern* and *output expression*. In the input pattern the local variable `a` is declared and assigned with the value of input token whenever action has just fired. The output expression uses that variable and send the value of `a` as a produced token to the output port at the end of the action firing.

Running the examples In the previously created project create new package net.sf.orcc.tutorial.l01SimpleActor

Create new CAL file named ID.cal and copy the following code there.

```
package net.sf.orcc.tutorial.l01SimpleActor;

actor ID () int In ==> int Out :
  first: action In: [a] ==>  Out: [a] end
end
```

In order to build working network we will need several additional actors to produce data and print results to the console.
For this example we will use `Source` and `Printer` actors as utilities. You can just download them from the Github repository and add to your project.

> `Source` actor is just a counter which produces a continuous sequence of numbers. You can also specify parameters for starting number (default is 1) and the counter upper bound (default is 10). To know how to do that see section *Other simple actors > Scale*

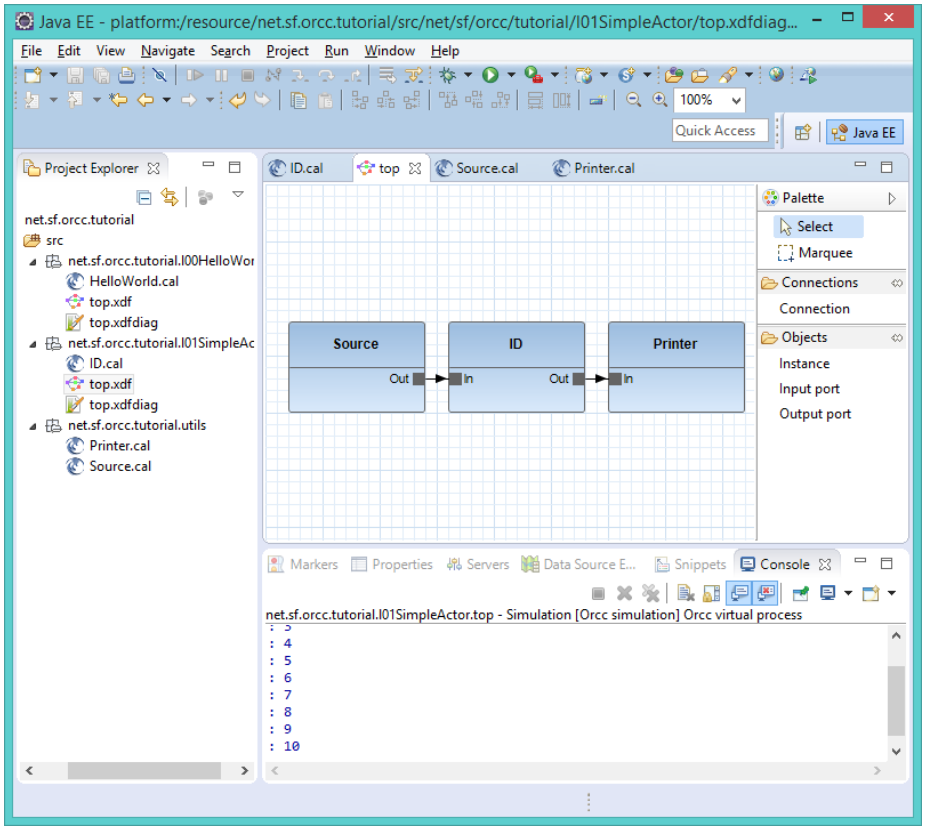
> `Printer` prints all the consumed tokens to the console. You can specify the name for each instance. That would be really useful if you have several of them printing to the same console simultaneously.

Now you can create the network just following the steps from previous lesson Hello world.

After you add instances of actors: `ID`, `Source` and `Printer`. Connect `Source` output to `ID` input, and `ID` output to `Printer` input.

> Hint: You can just drug-and-drop actor file from Project explorer pane to the XDF network diagram to add an instance of actor.

Now you can run example as was described in the previous lesson and see the result in the console.



Other simple actors

Add The next example shows how to make an actor which will be as simple as `ID` but at the same time will perform a real manipulation on the data

```
package net.sf.orcc.tutorial.l01SimpleActor;

actor Add () int In1, int In2 ==> int Out :
    action In1: [a], In2: [b] ==> Out: [a+b]
    end
end
```

We have now two *input ports* separated by comma `int In1, int In2` in the declaration of actor. Also *input pattern* changed to `In1: [a], In2: [b]` which means that action will *fire* only in case when both ports `In1` and `In2`

will have a valid data on their inputs. Consumed tokens than will be assigned to aand `b` respectively. Moreover, this example clarify the difference between input pattern and output expression. Looking at `Out: [a+b]` you can see that output expression includes real expression (sum of two variable), which will be calculated after action is finished; and result will be sent to the output port.

AddSeq Previous example consume two tokens from to input ports, but what happens if we have only one input port. Can we still add tow values? The following example provides the solution.

```
package net.sf.orcc.tutorial.l01SimpleActor;

actor AddSeq () int In1 ==> int Out :
    action In1: [a, b] ==> Out: [a+b]
    end
end
```

As you can see the input pattern `In1: [a, b]` consumes two tokens from the same input during one firing. You also can put more than two tokens separated by come in the input pattern.

AddSubSeq The output expression as illustrated in this example can also produce more than one token. You have just to write these expressions separated by coma within square brackets.

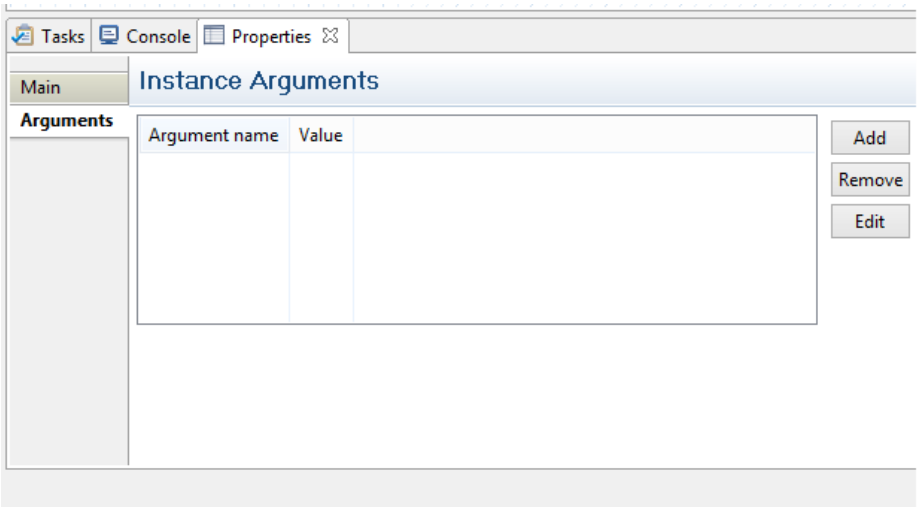
```
package net.sf.orcc.tutorial.l01SimpleActor;

actor AddSubSeq () int In1 ==> int Out :
    action In1: [a, b] ==> Out: [a+b,a-b]
    end
end
```

Scale Following example show another operation which can be performed by output expression.

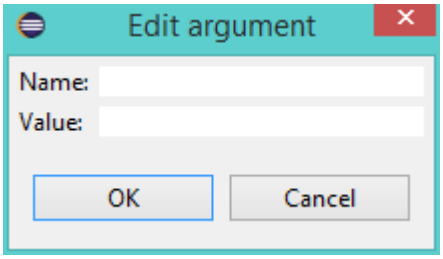
```
package net.sf.orcc.tutorial.l01SimpleActor;

actor Scale (int k = 1) int In ==> int Out :
    action In: [a] ==> Out: [k*a]
    end
end
```

You can notice that here we did not left the actor's parameters field empty. `int k = 1` introduce the parameter `k` which has the default value of `1`.

You can modify the parameters after you added an instance of actor to the XDF network. To do that you have to right-click on the instance rectangle and then chose *Show properties* item.



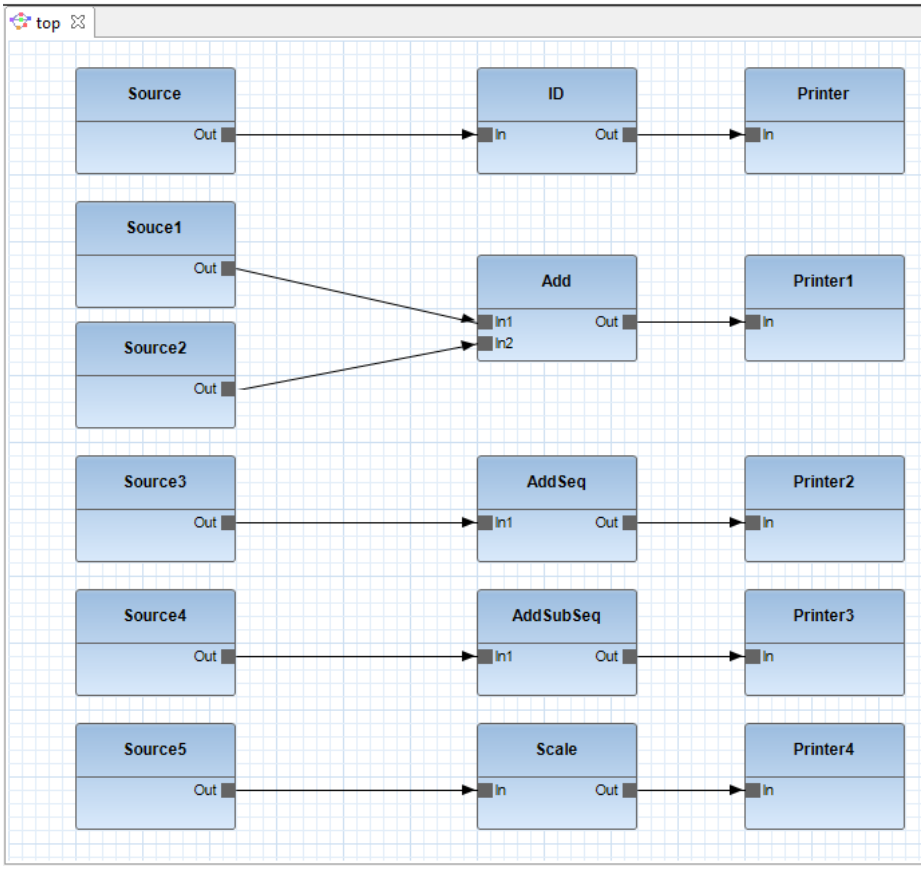
In the properties pane click on the *Arguments* on the left. You will see the list of arguments. Click on *Add* button and specify *Name* `k` and *Value* `7`.



Now when you run the network this particular instance of the actor will multiply the input token by 7.

> Hint: You can specify parameters for `Source` and `Printer` actors, which you downloaded from the Github repository in the beginning of the lesson.

Network of simple actors After you finish all the examples above you can build a network similar to the shown on the following picture.



Lesson 2. Nondeterminism

As was mentioned in previous lesson, actors may have multiple actions. So in the following example there are two.

```
package net.sf.orcc.tutorial.l02Nondeterminism;

actor NDMerge () int In1, int In2 ==> int Out :
    action In1: [x] ==> Out: [x] end
    action In2: [x] ==> Out: [x] end
end
```

This actor merges two input streams into one output. The first action takes a token from the input `In1` and sends it to the output `Out`. The second one does the same but for the input `In2`. However, from the description we cannot know

how actor will behave if there are tokens available on both input ports at the same time. The order of output tokens will be undefined. This behaviour is called *non-deterministic*.

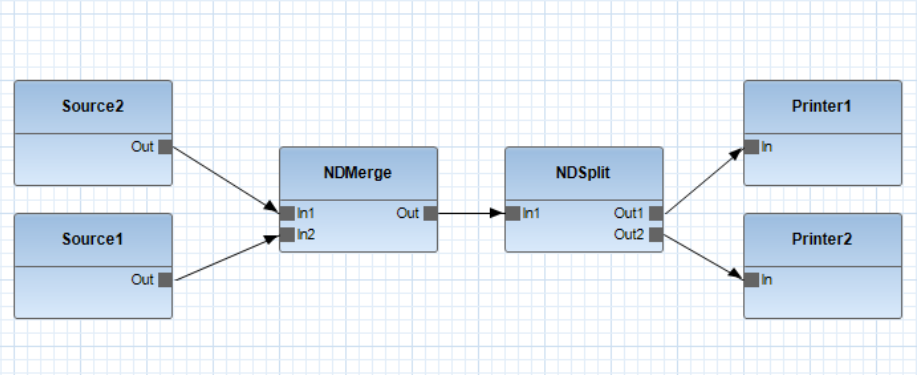
Generally, non-determinism means that the program can produce different output while processing the same input data. But in case of `NDMerge` the output is actually defined by the timings of the input streams. The ability to leave this choice open was added to the CAL language on purpose. For example, if there is no available data on the first stream, and there are data on the second stream, actor does not have to wait. It will send further the token from the input whichever will have it first. And if we know the timings of the input data, this actually will help to avoid stalls and unnecessary delays.

However, we can make an actor which will be really non-deterministic even if we know the timings of input data. The following example of `NDSplit` shows it.

```
package net.sf.orcc.tutorial.l02Nondeterminism;

actor NDSplit () int In1 ==> int Out1, int Out2 :
    action In1: [x] ==> Out1: [x] end
    action In1: [x] ==> Out2: [x] end
end
```

Here we have one input and two outputs. Two actions always have condition to fire at the same moment.

You can try to build the network similar to the following one to simulate the non-deterministic behaviour. Whilst, because of deterministic nature of simulator algorithms results will not look random.



Lesson 3. Guarded actions

In previous lesson we have introduced non-determinism of multiple actions within one actor. And even if we can exploit that property, surely in most of the cases

it's undesirable.

RVC CAL provides options to restrict action firing conditions. One of them is using *guards*. Is is a language construction which allows to specify additional requirements for action to fire.

In the following example you can see how *guards* can be used.

```
package net.sf.orcc.tutorial.l03GuardedActions;

actor Split () int In ==> int P, int N :

    action In: [a] ==> P: [a]
    guard a >= 0
    end

    action In: [a] ==> N: [a]
    guard a < 0
    end

end
```

Line `guard a >= 0` in the definition of first action defines a condition: so this action will fire only when the data on the input `In` will be greater or equal to zero. Similarly for the second action, `guard a < 0` means that it will fire only when data is less then zero.

It is important to notice that you are responsible for checking that the guard conditions of all actions within an actor are exhaustive, i.e. cover all possible input. Otherwise there will be cases when actor will stall forever.

Next example (which is wrong) shows what happens when you are not following this rule.

```
package net.sf.orcc.tutorial.l03GuardedActions;

actor SplitDead () int In ==> int P, int N :

    action In: [a] ==> P: [a]
    guard a > 0 end

    action In: [a] ==> N: [a]
    guard a < 0 end

end
```

Guard in the first action covers all the positive numbers, and guard in the second action covers all the negative ones. But what happens if we have zero on the

input `In`? This token won't cause any action to fire and (therefore) won't be consumed, so no other tokens will come to the input `In`. Actor will stall forever.

Moreover, besides being exhaustive *guards* in an actor should not have overlapped ranges. It can cause the errors explained in the following example.

```
package net.sf.orcc.tutorial.l03GuardedActions;

actor SplitND () int In ==> int P, int N :

    action In: [a] ==> P: [a]
    guard a >= 0 end

    action In: [a] ==> N: [a]
    guard a <= 0 end

end
```

Here we have two guards: one is `guard a >= 0` and another is `guard a <= 0`. So the first action fires when there is a non-negative number on the input `In`and the second action fires when there is a non-positive one. As you can notice, zero satisfy conditions for both. It means that in case of zero on the input we will have the same non-determinism problem which was described in the previous lesson.

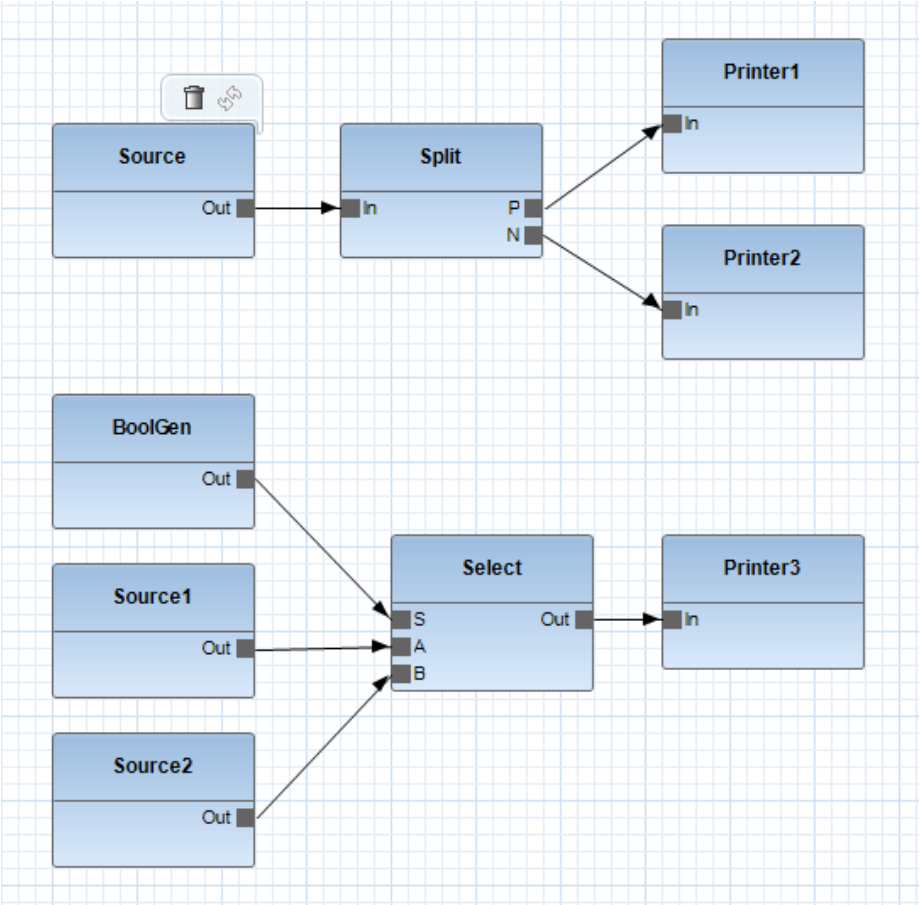
Final and important fact about the *guards* is that when guarding conditions is not fulfilled the action does not fire and *the token is not consumed* and remains on the input so it could be consumed by the next firing or another action. It can be illustrated in the following example.

```
package net.sf.orcc.tutorial.l03GuardedActions;

actor Select () bool S, int A, int B ==> int Out :

    action S: [sel], A: [x] ==> Out: [x]
    guard sel end

    action S: [sel], B: [x] ==> Out: [x]
    guard not sel end

end
```

The code above is similar to the `NDMerge` from the previous lesson, besides it has an additional input `bool S`, data from which is used to select action to fire. So in the example here, when we have tokens available on all three inputs, and token from input `S` is `false`, the first action *check it but does not consume it* so

the second action can consume it, fire, and send the data from input `A` to the output.

Now you can try to build network like the following. For the boolean input of the actor from the last example you will need special source generator BoolGen. You can download it from Github repository here. `BoolGen` actor generates an infinite sequence of `[true, false, true, false,...]`.



Lesson 4. State variables

So far, we saw only how actions can fire on not fire on external conditions, but there was nothing inside of actor which could affect subsequent firerings.

The *state variables* are internal memory of an actor. They represent actor's the internal state. Actions within an actor can change its internal state and thereby atlernate subsequent firings.

The simplest example of using state variable is a `Sum` actor.

```
package net.sf.orcc.tutorial.l04States;

actor Sum () int In ==> int Out :
    int sum := 0;
    action In: [a] ==> Out: [sum]
    do
        sum := sum + a;
    end
end
```

In line `int sum := 0;` we declare state variable `sum` and initialize it with zero. In this example you can also nitice that action can manipulate data within its body. In this case the code between `do` and `end` updates the state variable `sum` adding consumed token to it. Construction like that are usually accumulators. So here you can see that action not only cunsume input token and produce output, it also modifies internal state of the actor, which will affect the output of the next firing.

It is important to notice here (even thought it was mentioned in previous lessons) that the *output expression is evaluated after the action firing*. The value of `sum` in output expression `Out: [sum]` is the one which has been already updated by the action.

The previous example does not clearlly represent the *state meaning* of state variables. To explain that we will introduce the actor which selects the input stream according to internal state and send it the the output (recall `Select` actor from the previous lesson).

```
package net.sf.orcc.tutorial.l04States;

actor IterSelect () bool S, int A, int B ==> int Out :

    int state := 0;
    action S: [sel] ==> guard state = 0
    do
        if sel then
            state := 1;
        else
            state := 2;
        end
    end

    action A: [x] ==> Out: [x]
    guard state = 1
```

```
    do
        state := 0;
    end

    action B: [x] ==> Out: [x]
    guard state = 2
    do
        state := 0;
    end

end
```

Here in actor `IterSelect` we fisrt declare state variable `int state := 0;`.

The first action consume token from the input `S` and does not produce any output. It just modifies the internal state. (You can notice that there is no output expression after sign `==>`, but guard). So this action fires if current state is *zero* and changes it to `1` if there is `true` on the input `S` or to `2` if there is `false`.

The second action changes `state`to zero and copyies a token from input `A` to the output but only fires when internal state variable `state` *current* value is `1`. The third action do the same but only when `state` value is 2.

Note that `Select` and `IterSelect` are almost, but not entirely, equivalent. First of all, `IterSelect` makes twice as many steps in order to process the same number of tokens. Secondly, it actually reads, and therefore consumes, the S input token, irrespective of whether a matching data token is available on A or B. And unlike the previous examples, the `IterSelect` actor uses guards that depend on an actor state variable rather than on an input token.

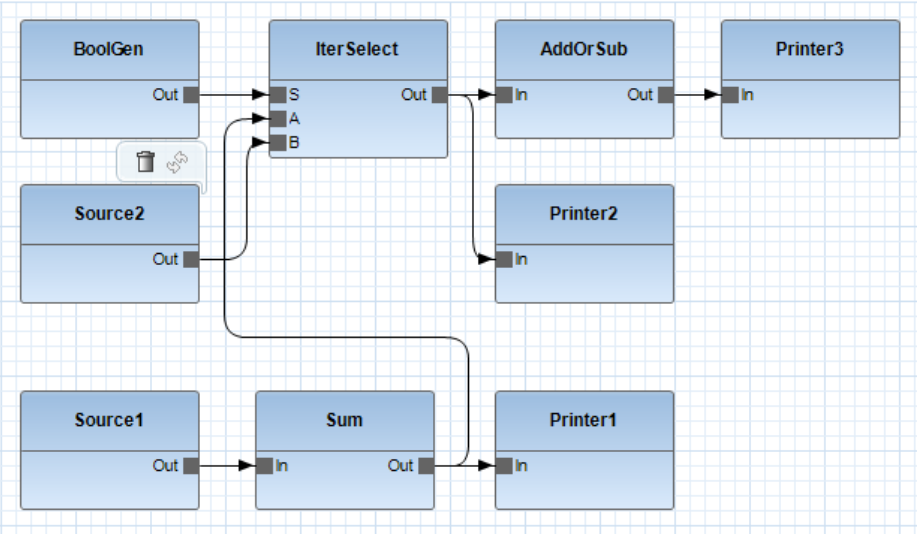
It is possible to use combinations of state variables and input tokens in guards, which is illustrated in the following example.

```
package net.sf.orcc.tutorial.l04States;

actor AddOrSub () int In ==> int Out :

    int sum := 0;

    action In: [a] ==> Out: [sum]
    guard a > sum
    do
        sum := sum + a;
    end

    action In: [a] ==> Out: [sum]
```

```
    guard a <= sum
    do
        sum := sum - a;
    end

end
```

Here we have to actions. One of them adds input token to the state variable `sum` and another substracts the input token from it depending on whether the token is less or not less then the value of `sum` itself.

You can build the network similar to the following diagram to experiment with these actors.



Lesson 5. Schedules

The `InterSelect` example in the previous lessons implements a commolny used software design pattern called *finite state machines* but describing it in that way is not very easy to understand.

RVC CAL provides special sintax to describe fitite state machines. It is called *schedules*. The following example `IterSelectFSM` illustrates using of *schedules*

```
package net.sf.orcc.tutorial.l05Schedules;

actor IterSelectFSM () bool S, int A, int B ==> int Out :
    readT: action S: [sel] ==> guard sel end
```

```
    readF: action S: [sel] ==> guard not sel end
    copyA: action A: [x] ==> Out: [x] end
    copyB: action B: [x] ==> Out: [x] end

    schedule fsm init :
        init (readT)  --> waitA;
        init (readF)  --> waitB;
        waitA (copyA) --> init;
        waitB (copyB) --> init;
    end
end
```

First you need to recall that every action can have identifier or lable, e.g. here `readT: action S: [sel] ==> guard sel end` the name of the action is `readT`. This lables are called *action tags*.

The block of code:

```
schedule fsm init :
    init (readT)  --> waitA;
    init (readF)  --> waitB;
    waitA (copyA) --> init;
    waitB (copyB) --> init;
end
```

describes our automaton. Basically, it is a textual representation of a finite state machine, given as a list of possible state transitions. The states of that finite state machine are the first and the last identifiers (`init`, `waitA` and `waitB`) in those transitions represented with sign `-->`. Relating this back to the original version of `IterSelect`, these states are the possible values of the state variable, i.e. `0`, `1`, and `2`. The initial state of the schedule is the one following `schedule fsm`. In this example, it is `init`.

Each state transition consists of three parts: the original state, a list of action tags, and the following state. For instance, in the transition `init (readT) --> waitA;` we have `init` as the original state, `readT` as the action tag, and `waitA` as the following state. The way to read this is that if the schedule is in state `init` and an action tagged with `readT` occurs, the schedule will subsequently be in state `waitA`.

The example above shows how we can make impementation simpler and more readable. But in fact, it complicicates the computation: in the original `IterSelect` actor we had only three actions and here we have them four.
Let's review a simpler example to learn how we can avoid incresing complexity using *schedules*.

Actor `AlmostFairMerge` merges two streams almost fair, as it is biased with respect to which input it starts reading from. But once it is running, it will strictly alternate between the two:

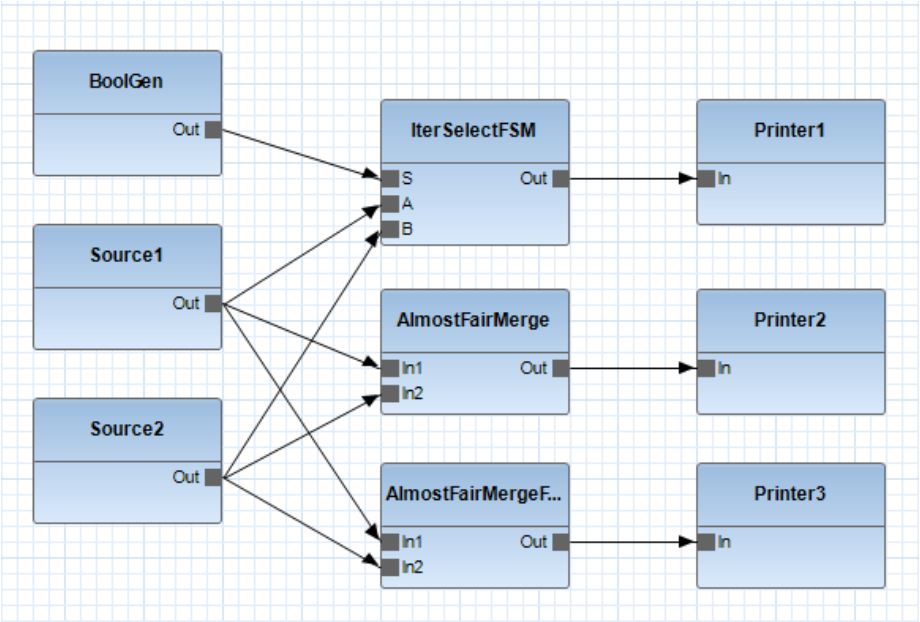
```
package net.sf.orcc.tutorial.l05Schedules;

actor AlmostFairMerge () int In1, int In2 ==> int Out :

    int state := 0;

    action In1: [x] ==> Out: [x]
    guard state = 0
    do
        state := 1;
    end

    action In2: [x] ==> Out: [x]
    guard state = 1
    do
        state := 0;
    end

end
```

The actor clearly has two states. So we will implement it using schedules:

```
package net.sf.orcc.tutorial.l05Schedules;

actor AlmostFairMergeFSM () int In1, int In2 ==> int Out :

    A: action In1: [x] ==> Out: [x] end
    B: action In2: [x] ==> Out: [x] end

    schedule fsm S1 :
        S1 (A) --> S2;
        S2 (B) --> S1;
    end

end
```

Here you can see two actions `A`, `B` and two states `S1`, `S2`, which is the same as in the original actor.

You can implement a network according to the diagram:



Lesson 6. Priorities

In the previous lessons we learnt about *guards*, *states* and *schedules*. But there is one more way to manage action firings in RVC CAL.

In case when contidions have met for more then one action to fire we can simple give higher priorities to some actions against others.

Following example explains how to use this in RVC CAL:

```
package net.sf.orcc.tutorial.l06Priorities;

actor BiasedMerge () int A, int B ==> int Out :

    InA: action A: [x] ==> Out: [x] end
    InB: action B: [x] ==> Out: [x] end

    priority
        InA > InB;
    end

end
```

Here we have two actions labeled `InA` and `InB`. And the line `InA > InB;` in the `priority ... end` block tells the actor that `InA` has a higher priority than

InB. So in case when tokens will be available on both inputs Aand B, the token from input A always goes to the output fist.

The following example illustrates how we can give equal priorities to groups of actions.

```
package net.sf.orcc.tutorial.l06Priorities;

actor FairMerge () int A, int B ==> int Out :

    One.a:  action A: [x] ==> Out: [x] end
    One.b:  action B: [x] ==> Out: [x] end
    Both.a: action A: [x], B: [y] ==> Out: [x,y] end
    Both.b: action A: [x], B: [y] ==> Out: [y,x] end

    priority
        Both > One;
    end

end
```

First you have to pay attention to the action tagging. We can group actions labeling them in the way One.a, One.b. So here we have a group One. Similarly, we tag othe two actions to the group Both.

And finally we give higher priority to the group Both.