ODOG

Framework for Programming Concurrent Platforms

http://odog.sourceforge.net/

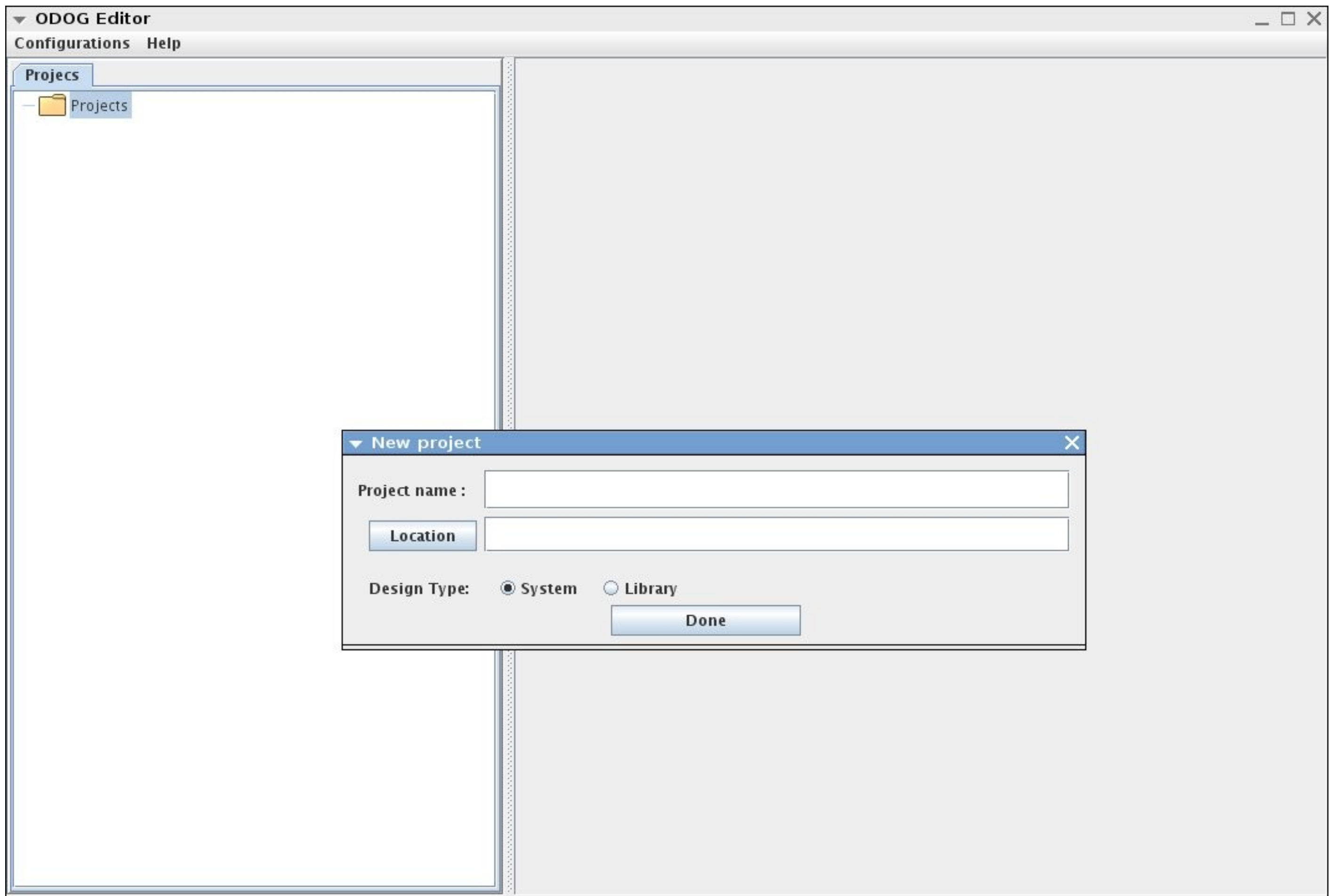
Version 1.0

February 2008

Copyright © 2006-2008 Ivan Jeukens

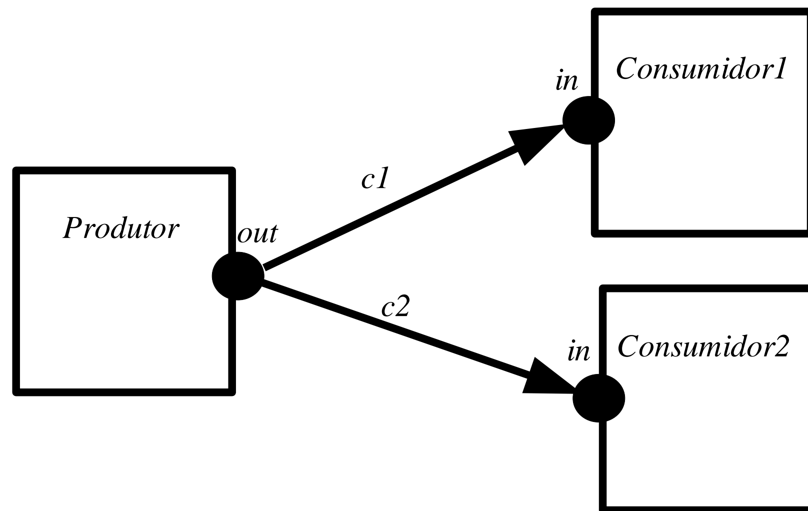
1 The Producer/Consumer

Start this demo by calling the *odogeditor* GUI editing tool. On the left frame, right-click on the **Projects** node. A pop-up menu appears with two options: **New Project** and **Import Project**. If an existing *xml* file describing a project exists together with all component descriptions, it can be imported within the editor using the second item of the menu. To start a fresh project, select the first option. A screen similar to the following should appear.



To create a new project, it is necessary to name it and so select a base directory where its data will reside. The new project must be either a System (specification that can be code generated) or a library (collection of artifacts). Name the project as *ProducerConsumer* and select any directory on your machine. The *ProducerConsumer* should appear under the *Project* tree node.

Now that the project is created, it should be populated with meta-artifacts and artifacts. A diagram of the system of this demo is shown on the next figure. Boxes represent components, circles data ports, and arrows connections.



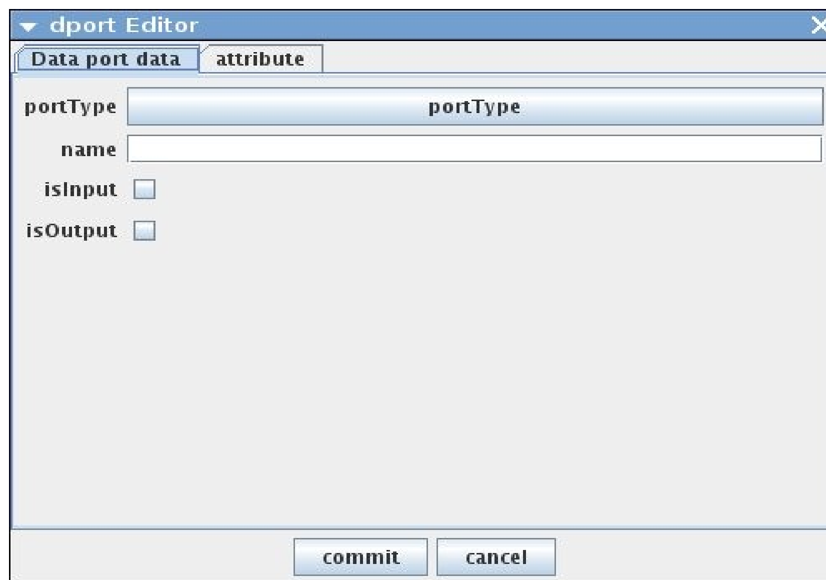
The system is composed of three components, where one produces data (*Produtor*) for the other two (*Consumidor1* and *Consumidor2*). These two components are instances of the same component. There are two distinct connections between the data ports: *c1* links output port *out* of component *Produtor* with input port *in* of *Consumidor1*, and likewise *c2* connections *Produtor* and *Consumidor2*.

In order to capture the above specification, three components will be necessary: an atomic component for *Produtor*, another atomic component for *Consumidor1* and *Consumidor2*, and a composite component for the topology show in the figure.

A right-click on the *ProducerConsumer* project tree node to reveal the options for manipulating a project. Click on **Add Artifact**. This should show the list of artifacts editors available within the framework. Choose **Atomic Component** and type **Producer** as the name of the component. Repeat this operation for the atomic component Consumer. For the composite component **top**, select the Composite Component editor. Now, all three components are created. You can save the project by using the **Save Project** option of the pop-up menu on ProducerConsumer tree node.

What has to be described now is the interface and behavior of each component. The first task is to add the syntactical elements. Right-click on the *Producer* component node in the projects tree and select the **edit** option. This should open on the right frame the respective artifact editor. A screen like this should appear:

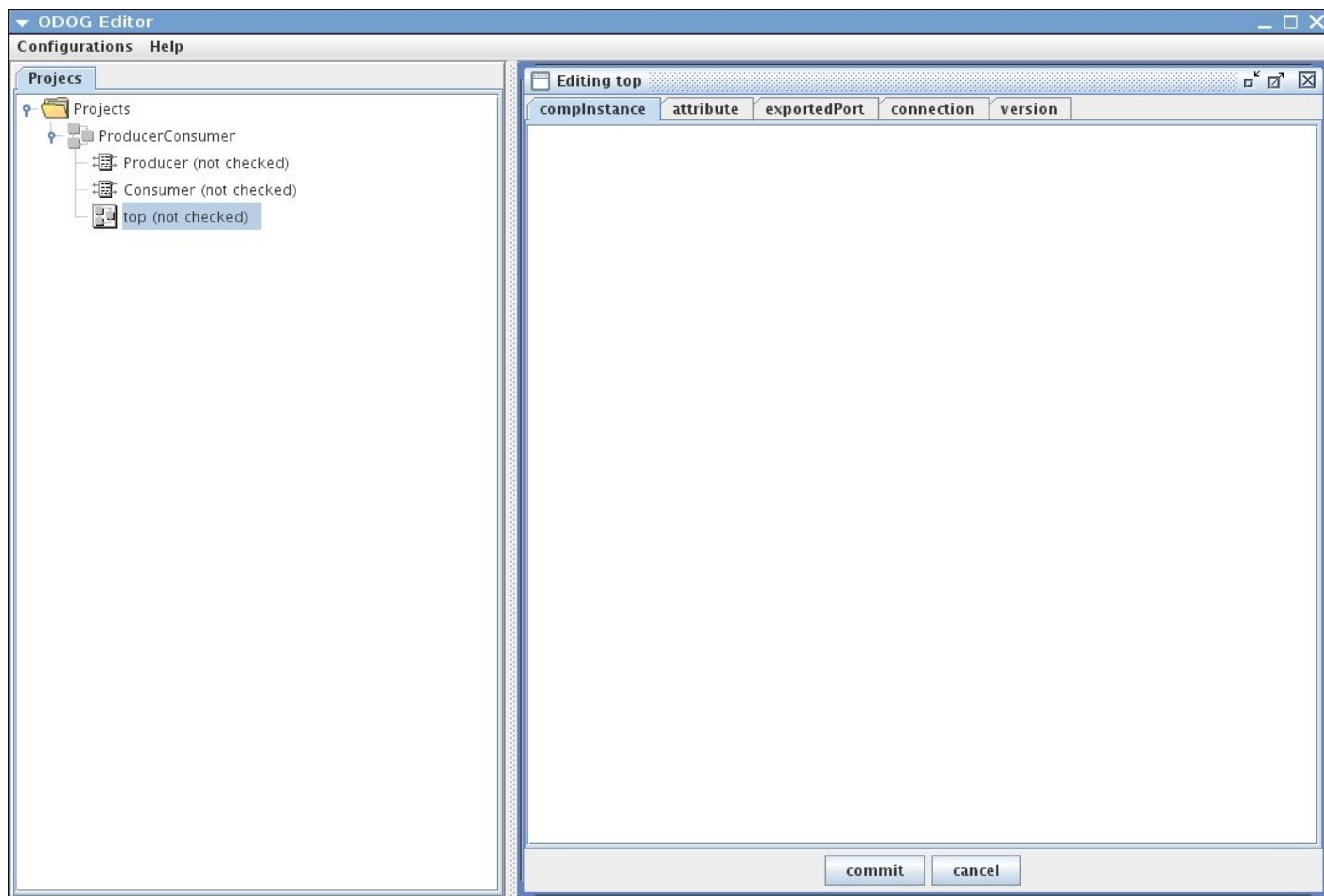
Both the atomic and composite artifact editors have a similar appearance. For each syntactical element, a list is associated. A context pop-up menu is available for each list, with three options: add, edit and delete. In the above screenshot, the lists for the interface elements (data port and attribute) and version element are shown. Right-click on the data port list and select the add option. The following dialog should appear:



Each syntactical element has a specific dialog. The above figure is the dialog for data ports of atomic components. It allows for the creation of such elements.

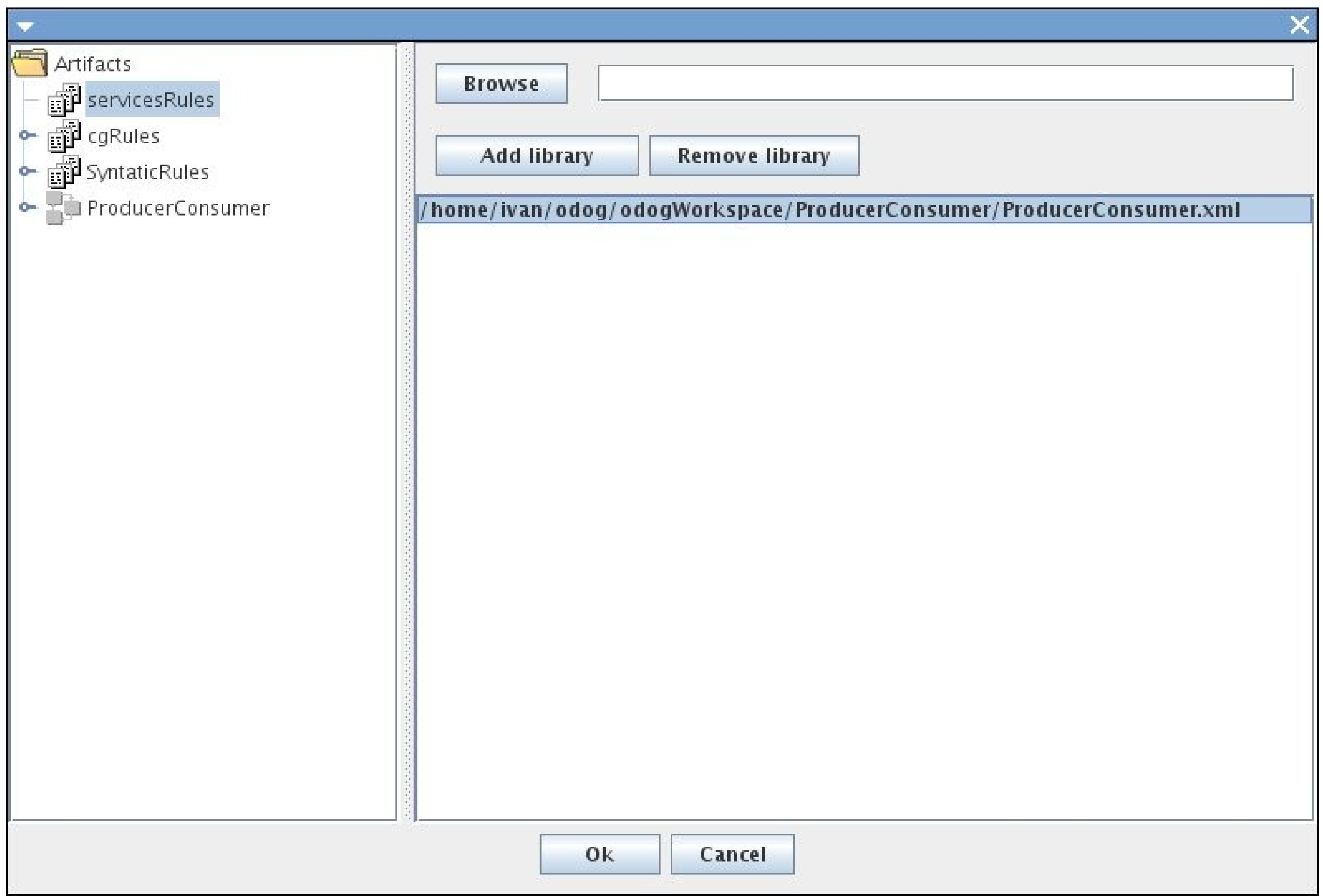
Create a port named **out**, select the **isOutput** option, and click on the portType button. This pops another dialog for editing the type of the port. Every port must have a type. Type in the valueExpr field the string "**char ***". This indicates for the code-generator the expected type flowing through this output port. Click on commit and again in commit. Now, the out port should appear on the list of data ports. Since the component *Producer* does not have any attributes, its interface is completed. However, in order to be instantiable, at least one version must be present. Create a new one clicking **add** on the context menu of list V**ersion**. The tool will prompt for a name of the version. Type **DEsim** and click OK. Click commit on the component editor frame. This finishes the editing of component *Producer*. Repeat the same operation for *Consumer* adding the **in** input data port and version DEsim.

So far we have two atomic components, with syntax and no behavior. At this point, one can choose to add behavior to then, or to create the syntax of the composite component. Lets do this last option. Click on the *top* component node and select edit. The composite component editor should appear:



Similar to the atomic component editor, the composite component editor first displays its interface elements together with the version element.

The specification requires one instance of the component *Producer* and two instances of the component *Consumer*. In order to add an instance, it is necessary to specify the component's name, and project where it is located. This project must be loaded within the editor. There are two methods for loading a project within the editor: the ODOG_LIBRARIES environment variable or the Library Index frame. To access it, on the menu **Configurations**, choose the item **Library Index**. This will show the following frame:



All the projects added via the ODOG_LIBRARIES variables will always be shown on the left frame. In this case, there is *SyntaticRules, cgRules* and *serviceRules*. The projects loaded into the GUI are automatically added to the Library Index. Therefore, the *ProducerConsumer* project can also be seen on the left frame. Its respective location is shown on the right frame. If one desires to add more projects to the Library Index, click the browse button and go to the directory where the project is located. After selecting the project's xml file, click **Add library**. This will include all the selected project's artifacts into the Library Index.

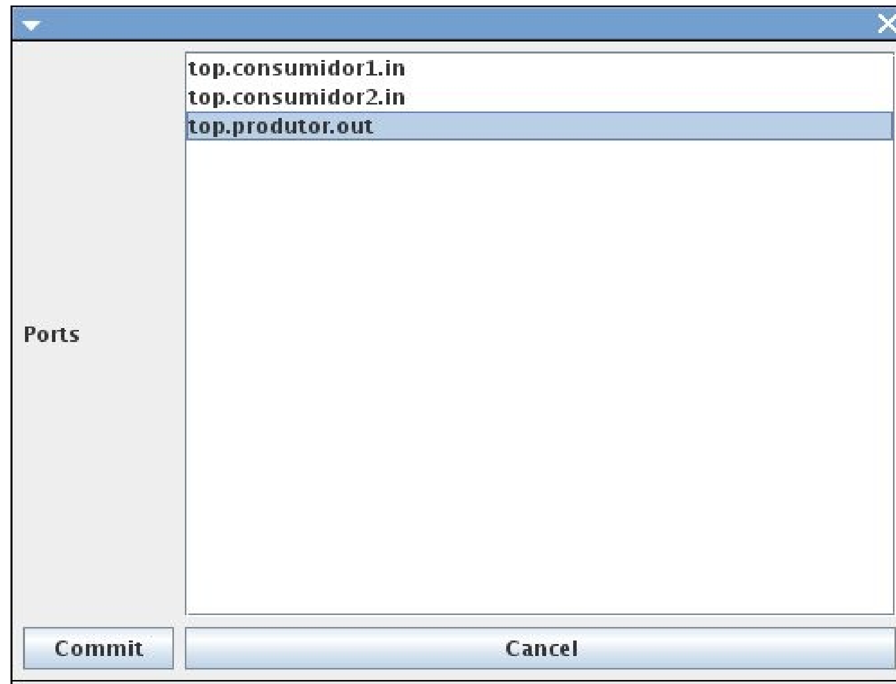
Click Ok and return to the composite component editor. Now the instances can be added. Select the component instance context pop-up menu, and click **add**. The following dialog will appear:

The intanceName field is a distinct name for the instance. Type *Produtor*. Then if you remember the name of the component and the name of the file describing the project where it resides, place them on the compName and libraryURL fields. Otherwise click on the View button and the Library Index frame will appear again. This time, instead of using the right frame for adding a project, navigate on the left frame, expanding the *ProducerConsumer* node. Select the *Producer* component and click Ok. You will see that the compName and libraryURL will be automatically filled with the correct values for the component that you just selected on the library frame. Click commit and the new component instance will appear on the compinstance list. Repeat this procedure for instances *Consumidor1* and *Consumidor2* of component *Consumer*.

Now that we have the necessary instances, the connections must be placed. Add a new connection by using the appropriate pop-up menu. The following dialog should be displayed:

The name field is a distinct name for the connection. Type *c1*. Output port is the data

sending port of this connection. Click on the button. The following dialog will appear:



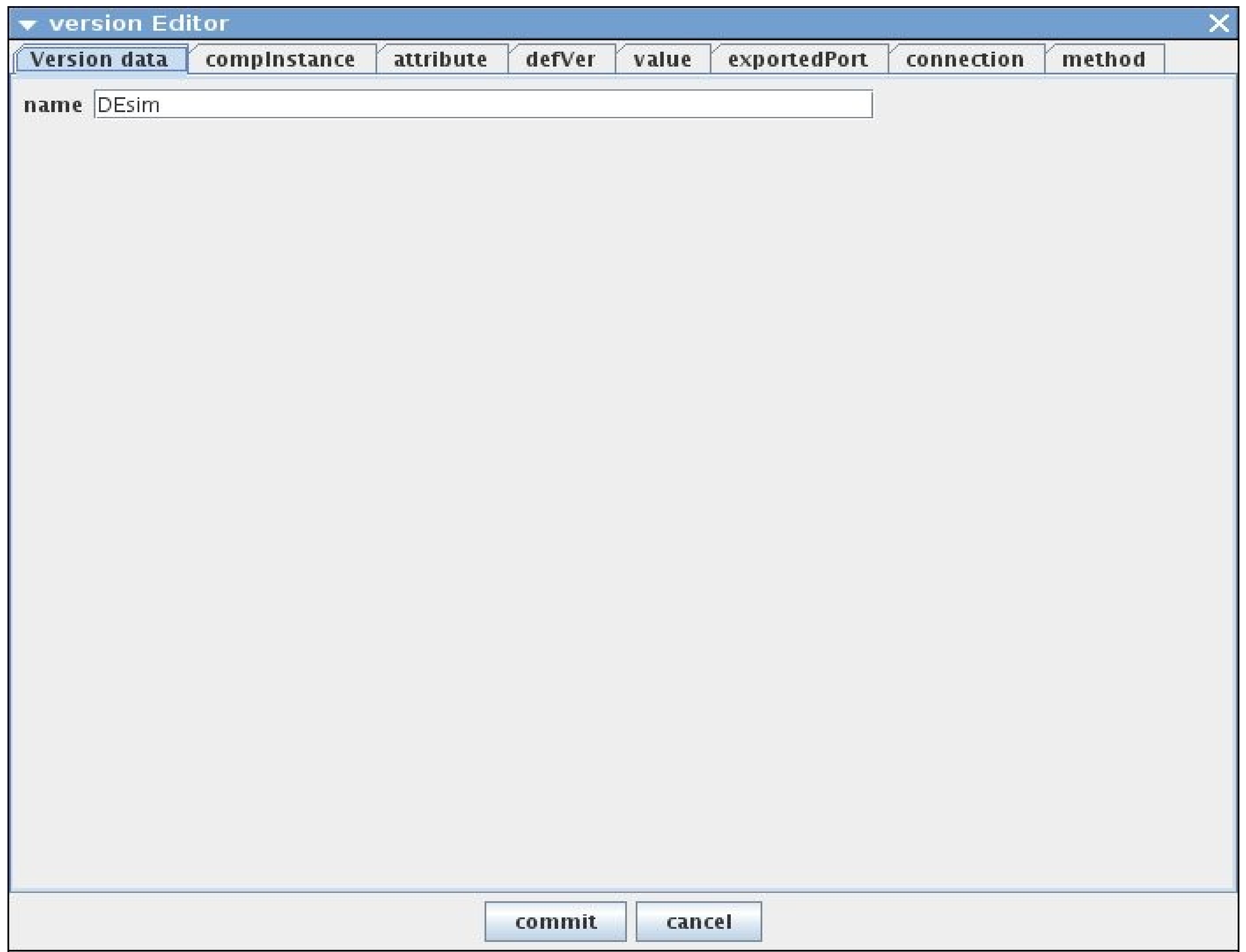
The Ports list shows the available ports that can be referenced. Since we are adding a connection at the interface level (not at version level), only ports of the interface of the available component instances are shown.

The names shown are full instance names (the name of the port plus the name of all syntactical elements up to the component containing it). Select the *top.produtor.out* port. Add the port *top.consumidor1.in* using the input port list. Click commit. Repeat this process for connection *c2*, but use the *top.consumidor2.in* as input port.

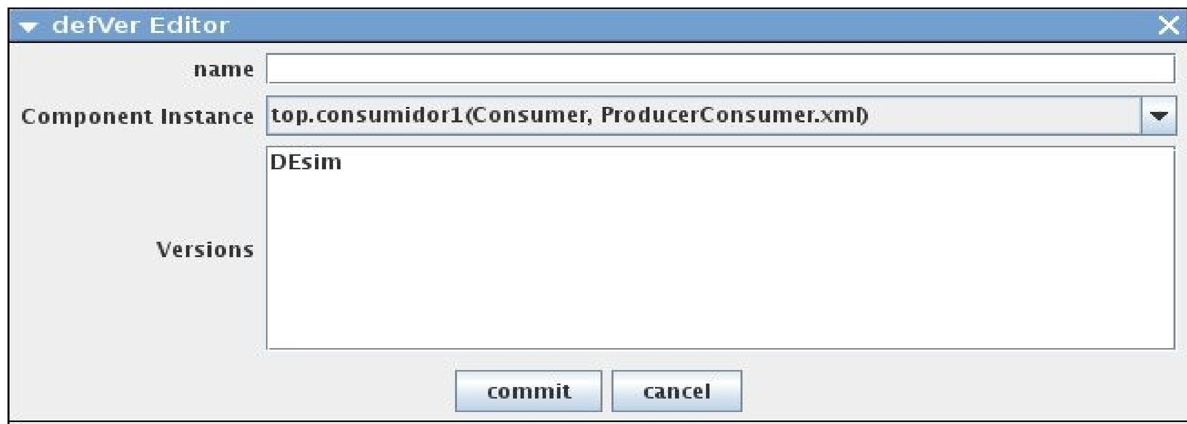
Next, if code generation is desired, two attributes will have to be added to some composite components: ISEM and Toplevel (these names are case-sensitive). The first one tells which interaction semantics to apply to the topology, and the second indicates that *top* is the toplevel component. The classifications of those attributes should be invisible, hasData and static for ISEM, and invisible no data for Toplevel. Go to the attribute tab, and click add on the pop-up menu. The following dialog will be shown:

First add the *Toplevel* attribute. For the ISEM attribute, when the classification is committed, the *has default value* check-box will be enabled. Don't and any default value.

Next a version must be added to the *top* component. Name it DEsim, and edit it by using the pop-up menu on DEsim. The following screen will be displayed:

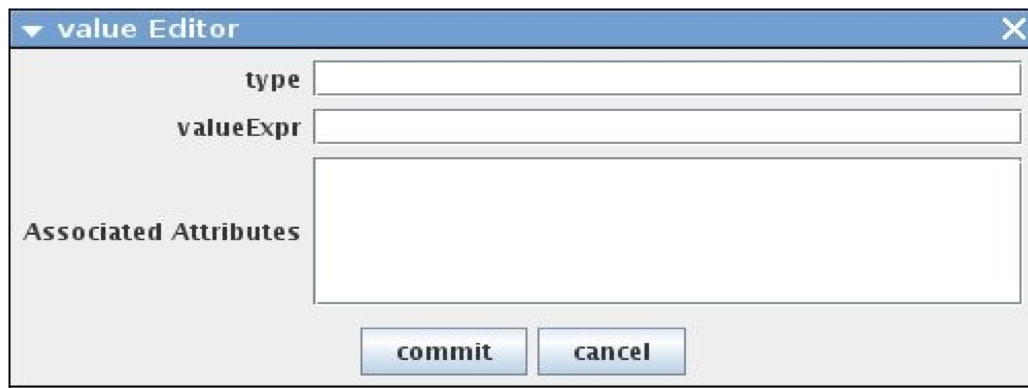


First, we need to associate a version for each component instance. Click the **add** option of the *defVer* element list, and the following dialog appears:



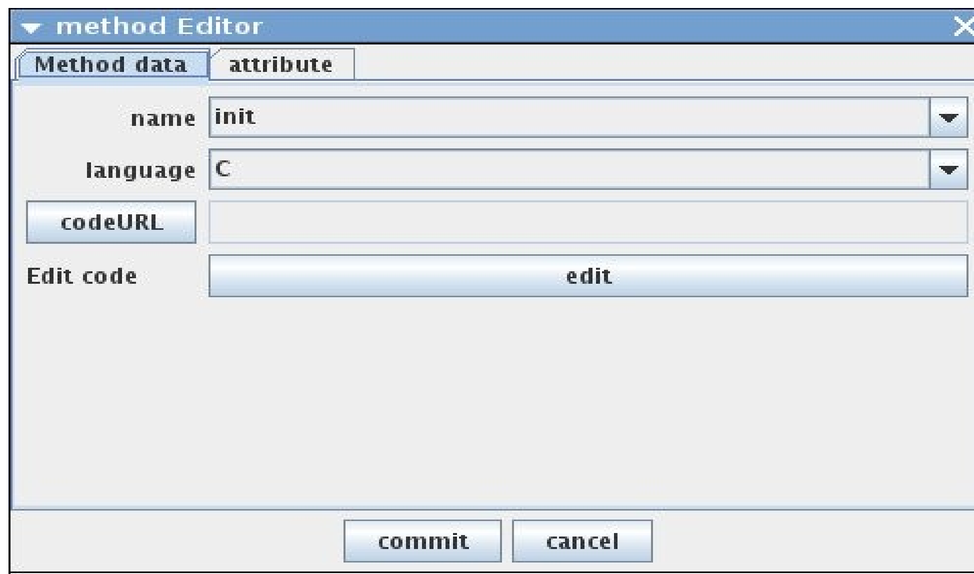
For each possible instance, the list of its available version is shown on the Versions list. Type any name, select a component and the DEsim version (the only option). This should be done for all three instances. Click commit for the version dialog.

Next, a value for the attribute ISEM should be created. Although a default value could be used (see the attribute dialog again), associating the value on the versions allows for different versions of this specification with different interaction semantics. Click on add for the value element. The value dialog will appear:



Fill the type text field with the string *char ** and for the valueExpr field the string “DE”. On the associated attributes list, click add option of the pop-up menu. This will show a list of available attributes. The attribute *top.ISEM* should appear. Selected it. Click commit.

So far, the interface for all components have been fully described. What remains to be done is the implementation of the behavior of the two atomic components. Edit version DEsim of *Producer*. Add a new method to it. The following dialog is displayed:



A method element is composed of the desired execution method (*init*, *compute*, *fixpoint* and *finish*), the desired host language, attributes if it is the case and the location, **within the directory of the project**, where the file containing the implementation code resides. The same file for more than one method can be used. For the DE ISEM, the *fixpoint* method is not defined. The edit button is supposed to open your favorite programming editor. Currently, it is hardwired to opening *jedit* (see jedit.sourceforge.net).

Three methods are necessary for component *Producer*. Place the following text in a file within the *ProducerConsumer* project directory. It contains the implementation of the three methods.

```
void init() {
    scheduleMe(1.0);
}

void compute() {
    if(currentTime() > 5.0) {
          char *str = “maio que 5.0\n”;
         sendDelayed("c1", str, strlen(str) + 1, currentTime() + 0.3);
    }
    else {
        char *str = “menor que 5.0\n”;
        sendAllDelayed("out", str, strlen(str) + 1, currentTime() + 0.35);
    }
    scheduleMe(currentTime() + 0.4);
}

#include <stdio.h>
void finish() {
    printf("Produtor encerrou\n");
}
```

The *init* method of *Producer* will schedule it for execution at time 1.0. This is necessary, since some component must be active at the beginning of execution, otherwise no event will be generated and the execution would end. Also, note that a component with no input ports is never executed, unless it schedules itself with the *scheduleMe* service. The *finish* method will generate a final message. The compute method will produce a string each time it is executed. Depending on the time instant of execution, it sends a string only on connection *c1*, or another string on all connections of port *out*. The *compute* method schedules the component for execution again after 0.4 time units.

Add the *init*, *compute* and *finish* method elements to the DEsim version, each pointing to the file with the above code. Click on the commit button. Now, the producer component is finished.

The *Consumer* component only needs the *compute* and *finish* method. Add those methods, pointing to the following code saved in some file:

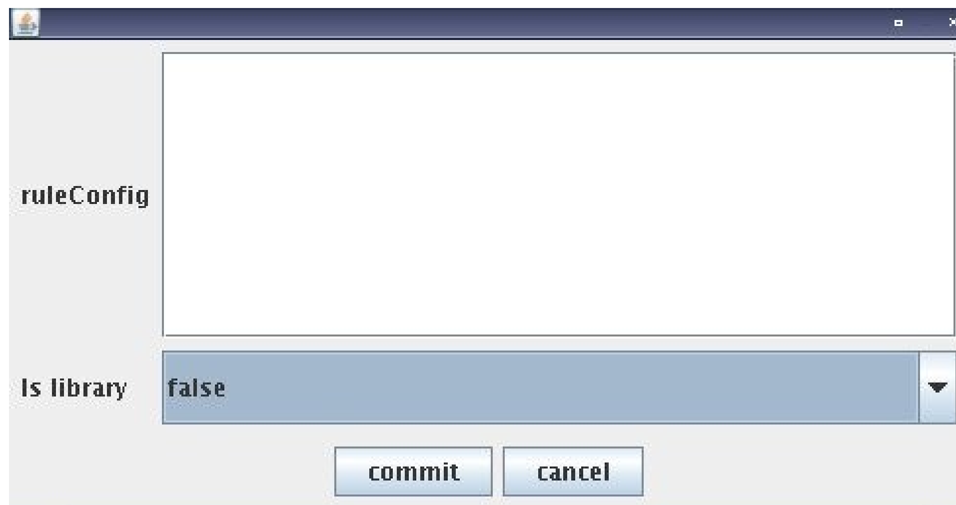
```
#include <stdio.h>
void compute() {
    char *data;
    int i, j;
    size_t vec;

      while(canReceiveAll("in", 1)) {
         int size = numberOfConnections("in");
         for(i = 0;i < size;i++) {
             receive(nameOfConnection("in", i), &data, &vec);
             printf("Recebi mensagem %s no timestamp %f\n", data,
                 currentTime());
         }
         free(data);
    }
}

void finish() {
    printf("Receptor encerrou\n");
}
```

The *Consumer* component will test if it can receive data from all connections associated with port *in*. Until there are new events, they will be read one by one and the message received displayed along with the value of time at the moment of execution.

Now the editing of components is complete. Before code for the specification can be generated, it must be checked for syntactic and semantic problems. Right-click on the *ProcuderConsumer* project node, and select **Configure**. The following dialog will appear:



The dialog shows the possible configurations for a project. Aside from altering its status from a project to a library, one can create rule configurations. A rule configuration is a list of rules that should be applied to a project. A rule configuration can contain used defined rules and system rules. The system rules are included by checking two check boxes. User rules are added selected individual rule artifacts from the Library Frame. Since the *ProducerConsumer* project does not have any extra rule, create a rule configuration, type a name for it, and apply the two check boxes for inclusion of system rules (syntactic and code generation). Click the commit buttons.

Having defined a rule configuration, we can check the project artifacts with it. Since the process of checking is done from the top hierarchy (toplevel composite component) down to the atomic components, it is only necessary to check from the topolevel component. Right-click on the *top* component tree and select the Check item. The result should be a Check Pass message. If there was any detectable flaw, a Check Fail message would be printed ,and the error message would be available. It is also possible to check the others artifacts, and during the process of development, it is advisable to do so.

Now, the last step can be done: code generation. Right-click on the *ProducerConsumer* tree node and select the **Generate Code** item. Had your project more than one toplevel component, the code generator would have asked to select one. Likewise if the selected toplevel component had more than one version. Since the *ProducerConsumer* has only one version and is the sole toplevel component, all is done without prompting.

There also must be a target platform for code generation. Select the *Host* platform for generating code for the host system. The result is a *Code generated successfully* message. Look at the directory where you placed the project. A subdirectory named *top.DEsim* is created. Go into it and you will see several .c files, plus the makefile for them. Type make,

and the binary **main** is created. If your code linked to external code, such as libraries, you would have to manually change the makefile.

When the DE model is used as a toplevel ISEM, it can accept to command line parameters: start time and stop time. They inform when to start and stop the simulation respectively. Type main 0 10 and the following output should be generated:

```
Recebi mensagem menor que 5.0 no timestamp 1.350000
Recebi mensagem menor que 5.0 no timestamp 1.350000
Recebi mensagem menor que 5.0 no timestamp 1.750000
Recebi mensagem menor que 5.0 no timestamp 1.750000
Recebi mensagem menor que 5.0 no timestamp 2.150000
Recebi mensagem menor que 5.0 no timestamp 2.150000
Recebi mensagem menor que 5.0 no timestamp 2.550000
Recebi mensagem menor que 5.0 no timestamp 2.550000
Recebi mensagem menor que 5.0 no timestamp 2.950000
Recebi mensagem menor que 5.0 no timestamp 2.950000
Recebi mensagem menor que 5.0 no timestamp 3.350000
Recebi mensagem menor que 5.0 no timestamp 3.350000
Recebi mensagem menor que 5.0 no timestamp 3.750000
Recebi mensagem menor que 5.0 no timestamp 3.750000
Recebi mensagem menor que 5.0 no timestamp 4.150000
Recebi mensagem menor que 5.0 no timestamp 4.150000
Recebi mensagem menor que 5.0 no timestamp 4.550000
Recebi mensagem menor que 5.0 no timestamp 4.550000
Recebi mensagem menor que 5.0 no timestamp 4.950000
Recebi mensagem menor que 5.0 no timestamp 4.950000
Recebi mensagem menor que 5.0 no timestamp 5.350000
Recebi mensagem menor que 5.0 no timestamp 5.350000
Recebi mensagem maio que 5.0 no timestamp 5.700000
Recebi mensagem maio que 5.0 no timestamp 6.100000
Recebi mensagem maio que 5.0 no timestamp 6.500000
Recebi mensagem maio que 5.0 no timestamp 6.900000
Recebi mensagem maio que 5.0 no timestamp 7.300000
Recebi mensagem maio que 5.0 no timestamp 7.700000
Recebi mensagem maio que 5.0 no timestamp 8.100000
Recebi mensagem maio que 5.0 no timestamp 8.500000
Recebi mensagem maio que 5.0 no timestamp 8.900000
Recebi mensagem maio que 5.0 no timestamp 9.300000
Recebi mensagem maio que 5.0 no timestamp 9.700000
Produtor encerrou
Receptor encerrou
Receptor encerrou
```

Type main 10 20 and this should be displayed:

```
Produtor encerrou
Receptor encerrou
Receptor encerrou
```

This is because the *Producer* is specified to schedule itself at 1.0, but with a start time of 10 this instant have already passed. The result is that it does not execute *compute*, does not

generate any event, and simulation is ended based on no events on the simulation queue.