



VerCors v4 Tutorial for the Users

INRIA Sophia-Antipolis

September 2016

oleksandra.kulankhina@gmail.com
eric.madelaine@inria.fr

VCE version 4.1.1

Contents

1	Introduction	2
1.1	VCEv4 Overview	2
1.2	Properties view	3
1.3	Installation	3
2	Architecture and Behaviour graphical specification	3
2.1	Example Overview	3
2.2	VCE project creation	4
2.3	Using an existing VCE Project	7
2.4	Building the component Architecture in VCE Component diagrams . . .	8
2.5	Types specification	8
2.6	UML Classes and Interfaces	9
2.7	UML Operations	11
2.8	Attach a UML Interface to a GCM Interface	11
2.9	Attach a UML Class to a GCM Component	12
2.10	Component attributes	12
2.11	Create and attach a State Machine to a UML Operation	13
2.12	Edit Local variables of a State Machine	14
2.13	Edit State Machines	14
3	Diagram Validation	17
3.1	Generating executable GCM/ProActive files	18
4	Generation of input for CADP	19
5	Reconfiguration	20
5.1	Graphical design	20

1 Introduction

VerCors is a platform for the specification, analysis, verification and validation of the GCM- based applications architecture and behaviour. It contains a set of tools including VCE v.4. VCE v.4 is a graphical designer for the GCM architecture and Components behaviour specification. It is distributed as a set of Eclipse plug-ins.

This guide explains the basic functionality of VCE v.4. It provides step by step instructions for creating a simple example of a Component model using the VCE editors, including all facets of this model: classes and interfaces, types, architecture, and behaviours. It also explains how to validate the static semantics properties of the Component models, how to produce (partial) executable code in GCM/ProActive and how to produce models that can be verified by CADP¹ model-checker.

Additional information:

VCE v.4 is based on Sirius² technology and on the functionality of the standard editors created with Sirius. Hence, if one wants to get more specific/advanced information that would not be found in this tutorial, it is recommended to read some of the Sirius documentations. In particular:

- Getting started for End-users: <https://wiki.eclipse.org/Sirius/Tutorials/4MinTutorial>
- Sirius user manual: <http://www.eclipse.org/sirius/doc/user/Sirius%20User%20Manual.html>

1.1 VCEv4 Overview

As in all Eclipse-based environments, all models and diagrams related to a given application are grouped in a Project.

A **VCE Modeling project** contains models (**VCE** models for the architecture description, **UML** classes and interfaces, UML State Machines for behaviour description, and **VceTypes** models for types definition). A VCE Modeling project also includes diagrams (VCE Components diagrams, UML classes, interfaces, and state-machine diagrams and a VCE Types diagram) which illustrate the elements of the models. A VCE model has links to UML models. In particular, a GCM interface refers to a UML Interface, a GCM Primitive component refers to a UML Class. The UML Operations and State Machines use VCE Types for the typing of the arguments and return values.

VCEv4 mainly contains three modules:

- Graphical editors the design of component architecture and behaviour;
- A module translating graphical models into Executable Java/Proactive code;
- A module translating graphical models into input for CADP model-checker.

¹<http://cadp/>

²<http://www.eclipse.org/sirius/>

1.2 Properties view

You will often need to edit some parameters of your model in Eclipse **Properties** view. If you do not see it in your workspace, in order to open it you should:

1. Click on **Window** on the top-bar menu of Eclipse (Obeo) and select **Show view** → **Properties**.

1.3 Installation

In order to install VerCors, the user should, first, install Obeo Designer. For this, it is necessary to download and unzip the Obeo Designer Community 8.x version from <https://www.obeodesigner.com/en/download>. Then, the user can use the Eclipse software installation manager³ to install VCEv4 on Obeo Designer from VCEv4 repository: <http://vercors.gforge.inria.fr/repository/>.

2 Architecture and Behaviour graphical specification

In this section we explain how to:

- create a new VCE Project or import it from an existing source;
- edit GCM Component diagrams; specify types;
- create and edit UML classes, interfaces, operations and attributes;
- connect the operations of UML classes and interfaces;
- attach a UML interface to a GCM interface;
- attach a UML class to a GCM primitive component;
- specify default values of UML attributes;
- create and edit a State Machine and attach it to a UML operation.

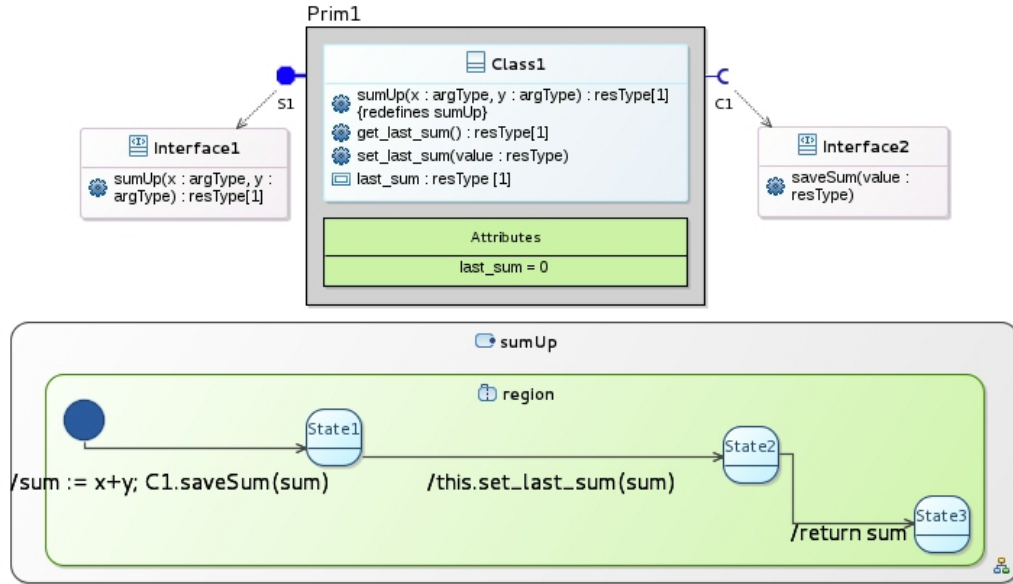
We will illustrate the use of graphical editors with an example described in the section below.

2.1 Example Overview

A figure below illustrates the example. This is a very simple VCE Component diagram containing one primitive component **Prim1** with one server and one client GCM interface (**S1** and **C1**). Each GCM interface is connected to a UML interface (**Interface1** and **Interface2** correspondingly). Interface1 has an operation **sumUp** that can be served by the component, Interface2 has the list of the operations that can be called by the

³<http://help.eclipse.org/luna/index.jsp?topic=/org.eclipse.platform.doc.user/tasks/tasks-124.htm>

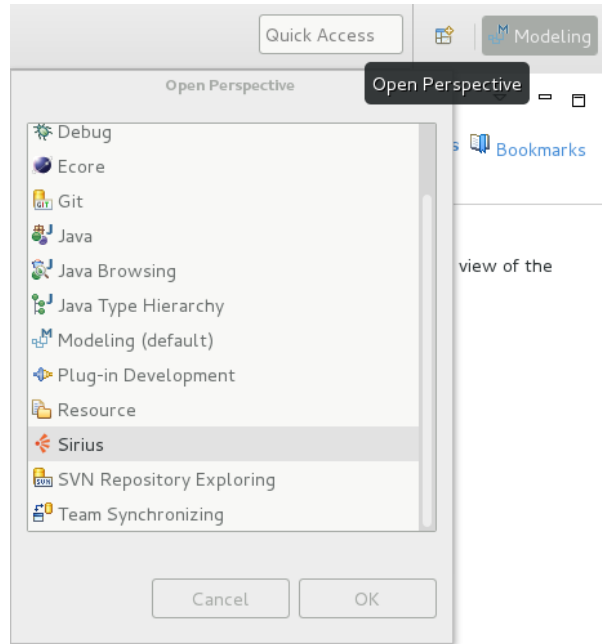
component. A UML class **Class1** is attached to the component. It has one method implementing the operation of the server interface (**sumUp()**), one attribute **last_sum** and two methods **get_last_sum()** and **set_last_sum()** to access the value of the attribute. The behaviour of the sumUp method is illustrated on a UML State Machine **sumUp**. The default value of attribute **last_sum** is given in a green box and is equal to 0.



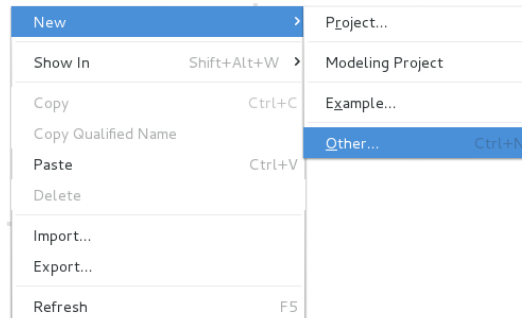
Prim1 implements the following business logic. Method **sumUp** can be called on the server interface **S1**. It takes two arguments: **x** and **y**, computes their sum, saves the sum to its local attribute **last_sum**, sends the sum to another component through its client interface **C1** and returns the sum to the caller.

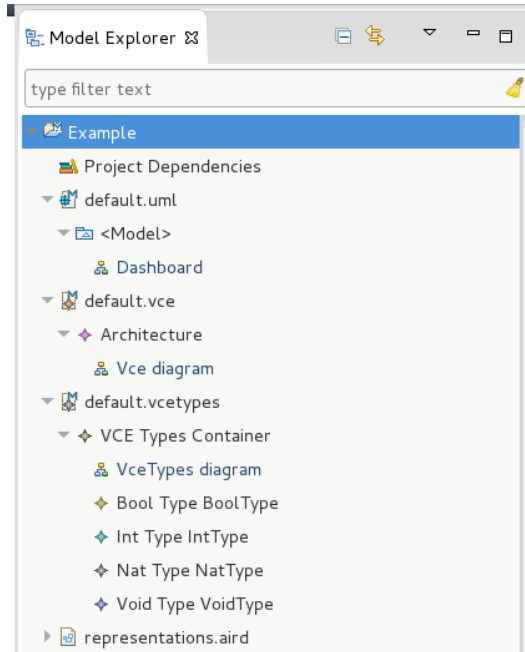
2.2 VCE project creation

1. Go to **Sirius** perspective (see the figure below). You can change the perspective using a button in top-right corner of your Obeo (Eclipse) or in the top-bar menu.

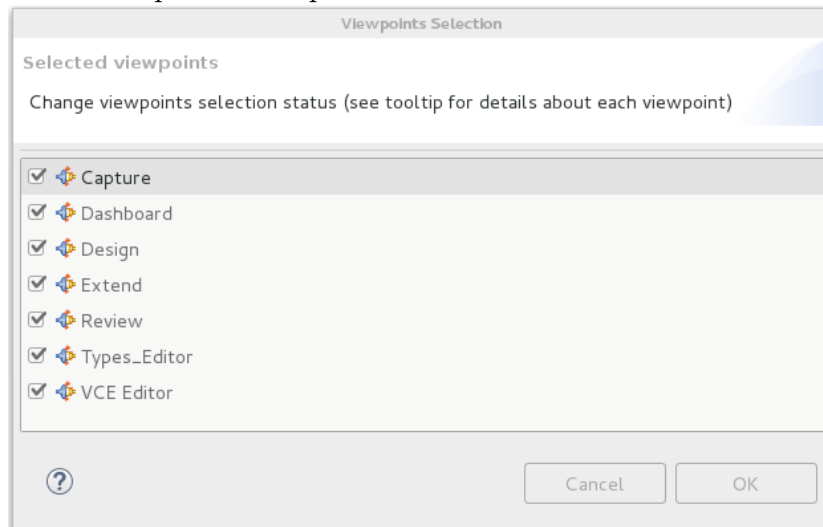


2. Create a new VCE project: right-click in the Model Explorer and select **New**→**Other...** (see a figure below). Then, select **VCE Wizard**→**VCE Project** and hit **Next**.

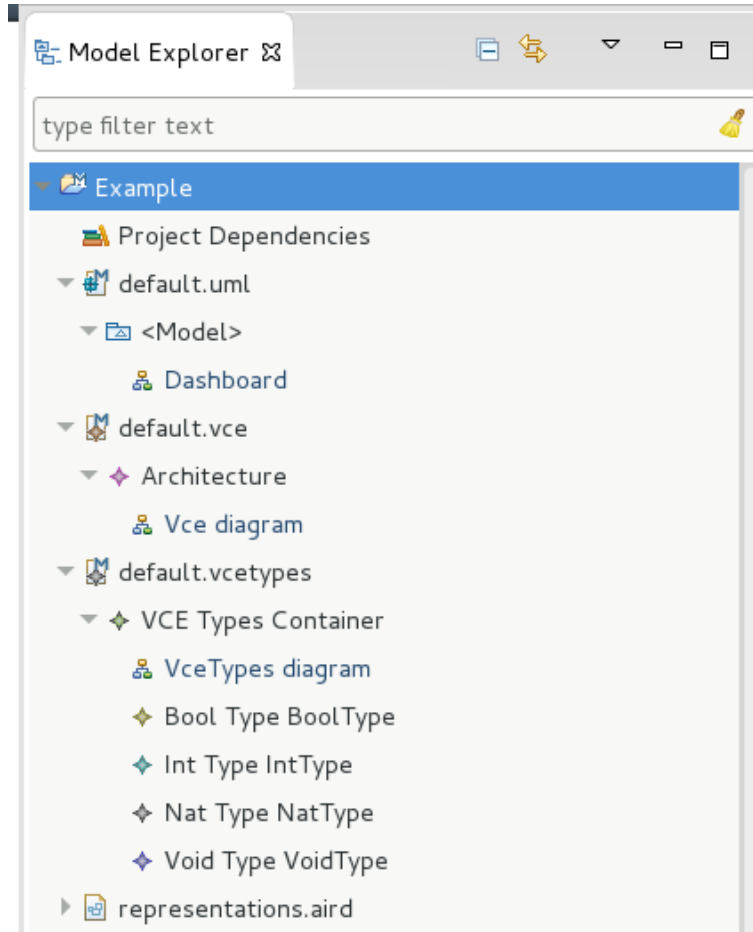




3. Give a name to the project and press on **Finish**;
4. Right-click on the created project in Model explorer and select **Viewpoint selection**.
5. Check all the viewpoints and press Ok.



As a result, a new VCE project is created. Its structure is illustrated at the figure below.



A VCE project has the following elements:

- default.uml – a UML model;
- Dashboard – can be used to create diagrams;
- default.vce – a VCE model;
- VCE diagram – a diagram illustrating elements of default.vce;
- default.vcetypes – a model containing the types specification;
- VceTypes diagram – a diagram illustrating elements of default.vcetypes.

2.3 Using an existing VCE Project

When working in collaborative environments, you will often have to share projects with your colleagues. You may also have to download an existing project. Importing such a project in your VCE workspace works as for any other Eclipse project:

1. import the archive file (e.g. zip file), and save it somewhere on your file system;
2. from the **File** menu, select **Import**;

3. select **General** → **Existing project in workspace** then hit Next;
4. browse to find your project, hit Ok;
5. then hit Finish.

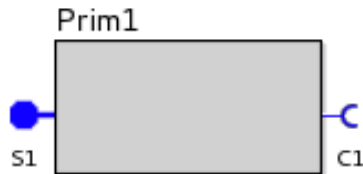
The corresponding project will appear in the Model explorer panel, with a structure similar to the structure created by the VCE wizard, though of course there will usually many be more elements in there.

2.4 Building the component Architecture in VCE Component diagrams

This section describes how to edit GCM-based application architecture in VCE Component diagrams.

1. Normally, a Component diagram should be created in a VCE project by default. In order to create a new diagram right-click on **Architecture** and chose **New Representation** → **new Component Diagram**.
2. You can edit it using tools on the Palette. You can also use a toolbar on the top of a diagram editor. You can change the properties of your model elements using Properties View.

For our first example pick the tool called **Primitive** and draw a primitive component. Then, add to the component two interfaces using **Server** and **Client** tools. You can edit the names either directly on the diagram or in the Properties view. The resulting diagram is illustrated at the figure below.



2.5 Types specification

This section describes how to specify types. VCE has 4 predefined type: VoidType, BoolType, IntType and NatType. However, one can construct his own types: enumerations, records, arrays, and integer intervals to abstract the data domains. Our example has two integer intervals: **argType = 0..2** and **resType = 0..4**.

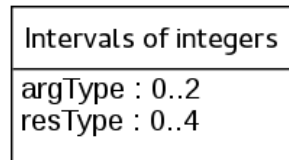
In order to specify the types, you need to:

1. Open an existing VceTypes diagram or create a new one. In order to create a new diagram right-click on **VCE Types Container** in Model explorer and choose **new Representation** → **new Types Diagram**.

2. You can edit it using tools on the Palette.

Note: in the current version of VCE most of the features of the types (e.g. the name of a type) can be edited in **Properties** view but not directly on a diagram.

For our first example pick the tool called **Intervals Container** and draw a container where you will specify the intervals. Then, using **Int Interval** tool add two intervals. Use Properties view to set their names, lower and upper bounds. The result is illustrated below.

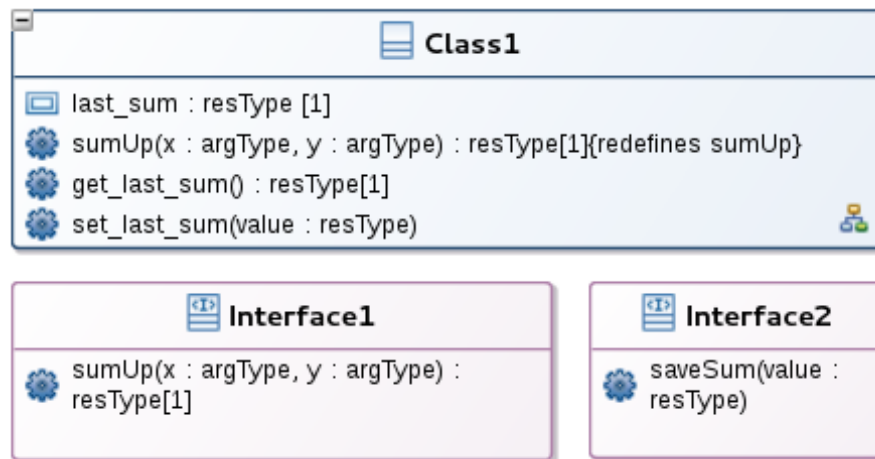


2.6 UML Classes and Interfaces

The UML Classes, Interfaces, and Operations that will be used to specify GCM interfaces and primitive component implementation class must be described at UML Class diagram. In order to create it, right-click on **<Model>** in the Model explorer and select **New Representation → Class diagram**. A new diagram will be created and opened automatically. Another way is to double-click on **Dashboard** and there one can find Class diagram.

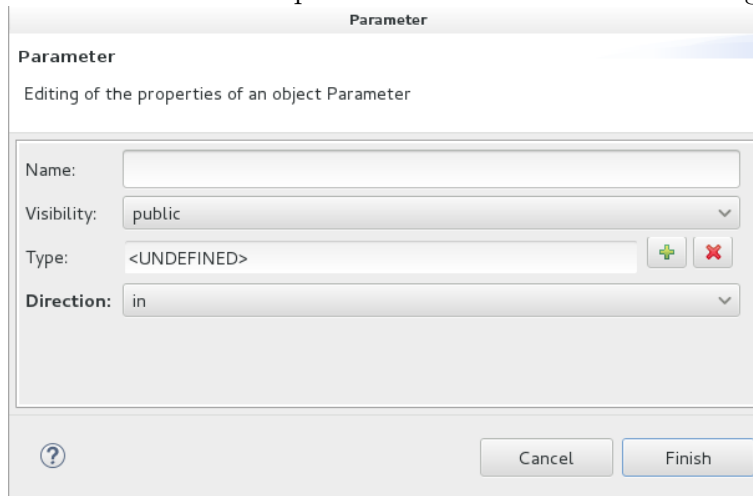
[Hint: choose meaningful, but reasonably short names. Long names make graphical diagrams difficult to manage]

The figure below illustrates Class diagram of our example.



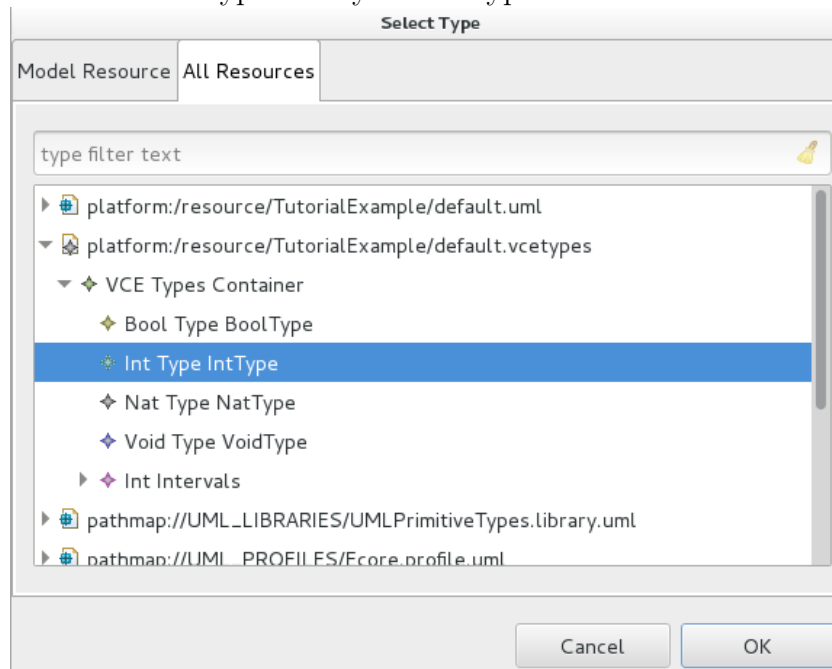
It has two UML Interfaces: **Interface1** and **Interface2**, and one UML Class **Class1**. In order to create them, **Interface** (resp. **Class**) tools are used. The tool **Operation** is used in order to add an Operation to a Class or an Interface (pick-up the tool and click on the bottom part of a the Class or an Interface where you want to add the Operation). You can modify the Operation in its Properties view. The name can be set in **General** tab. The specification of the arguments and return type is a little bit more tricky. In order to modify them, you need to:

1. Go to the Parameters tab of an Operation Properties view.
2. Click on the **green plus** button in the top-right corner of the Properties view; this will open a window to create a parameter as illustrated on the figure below.



The image shows a dialog box titled "Parameter". Below the title bar, it says "Parameter" and "Editing of the properties of an object Parameter". The dialog contains four fields: "Name:" with an empty text box, "Visibility:" with a dropdown menu showing "public", "Type:" with a dropdown menu showing "<UNDEFINED>" and two small buttons (a green plus and a red X) to its right, and "Direction:" with a dropdown menu showing "in". At the bottom, there is a question mark icon, a "Cancel" button, and a "Finish" button.

3. The window includes fields for specification of the name, type and direction of the parameter. You parameter can be either an argument of an operation, or its return type. Enter the name if you are specifying an argument; select the direction: *in* for the arguments or *return* for the return type. You should also select the parameter type among the ones declared in the VceTypes model. In order to specify type, click on **green plus** button next to the field Type, go to **All Resources** tab and select the needed VceType from your .vcetypes model. See an example below.



The image shows a dialog box titled "Select Type". It has two tabs: "Model Resource" and "All Resources", with "All Resources" currently selected. Below the tabs is a search bar labeled "type filter text". The main area is a tree view showing a hierarchy of resources. The tree structure is as follows:

- platform:/resource/TutorialExample/default.uml
 - platform:/resource/TutorialExample/default.vcetypes
 - VCE Types Container
 - Bool Type BoolType
 - Int Type IntType** (highlighted)
 - Nat Type NatType
 - Void Type VoidType
 - Int Intervals
- pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml
- pathmap://UML_PROFILES/Ecore.profile.uml

 At the bottom, there are "Cancel" and "OK" buttons.

An example of **x** argument of **sumUp** operation is given on the figure below.

An attribute (or property) of a class can be specified using a tool **Property**. Among the properties of an attribute the most important ones are its name, type and default value.

2.7 UML Operations

We should distinguish three kinds of UML Class Operations:

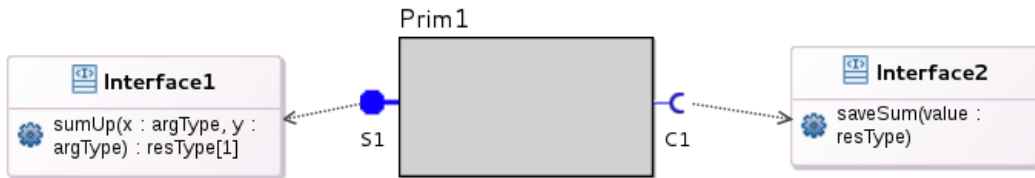
- **Server operations.** These are the operations that will be accessed from outside of a component. They implement the behaviour of server interfaces operations. In order to establish the relation between a server operation of a class and the implemented method of an interface you need to do the following. (1) go to the Properties view of your class operation and open **Semantic** tab. (2) in the field **Redefined operation** select the implemented operation of an interface.
- **Local operations.** These are class operations that are used for local computations and are not accessible from outside of the component.
- **Get and set operations.** This is a special kind of local operations that are used inside a component to get or set value of its attributes. One get and one set operation should be manually declared for *every attribute of a Class*. The signature of a get operation is `get_attrName():attrType`; the signature of a set operation is `set_attrName(value:attrType)`.

2.8 Attach a UML Interface to a GCM Interface

In our example, a UML Interface Interface1 is attached to a GCM Interface S1 and Interface2 is attached to C1. Which means that operations of Interface1 can be called on S1 and operations of Interface2 can be called from C1. In order to attach a UML Interface to a GCM Interface you need to:

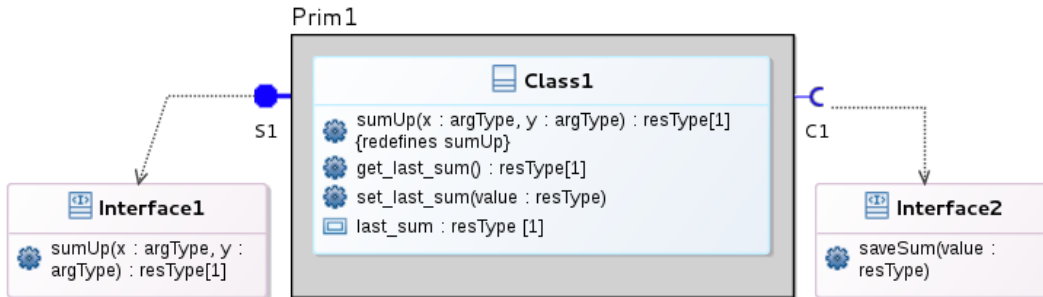
1. Open the VCE Components Diagram.
2. Right-click on the GCM interface to which you want to attach a UML Interface (S1 or C1 in our example).
3. Select **Attach UML Interface** option.
4. Select the required UML interface from the given list (Interface1 or Interface2 in our example)

In the case of our example if you specify everything correctly, you should get the diagram illustrated below.



2.9 Attach a UML Class to a GCM Component

UML classes illustrate the lists of the operations that can be processed by GCM primitive components and the lists of possessed attributes. In our example, Class1 contains the methods of a component Prim1. In order to specify the relation between a primitive and its implementation class, right-click on the primitive component and select **Attach UML Class** option. Then, select the required class from the given list. The result is illustrated on the figure below.



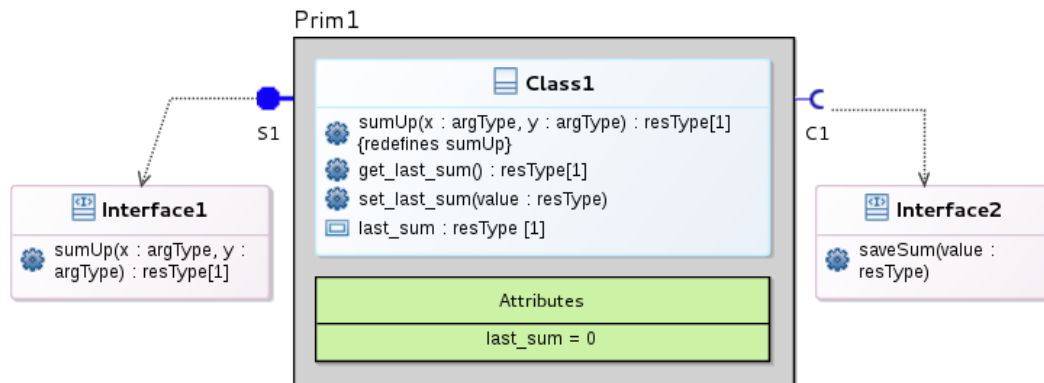
2.10 Component attributes

A UML class attached to a primitive component defines the list of the component's attributes. The default value of an attribute can be always set on the UML Class Diagram. However, it is often needed to set the default value of an attribute in the context of a concrete primitive component. In order to do so you should:

1. Open VCE Component Diagram.
2. Pick **Attributes specification** tool and add a box for attributes specification to the primitive component.

3. Use tool **Attribute value** to add attributes that have specific default value to the attributes specification box.
4. You can set the name and the value of an attribute in Properties view. **Note:** the name of the attribute should strictly correspond to the one on a Class diagram.

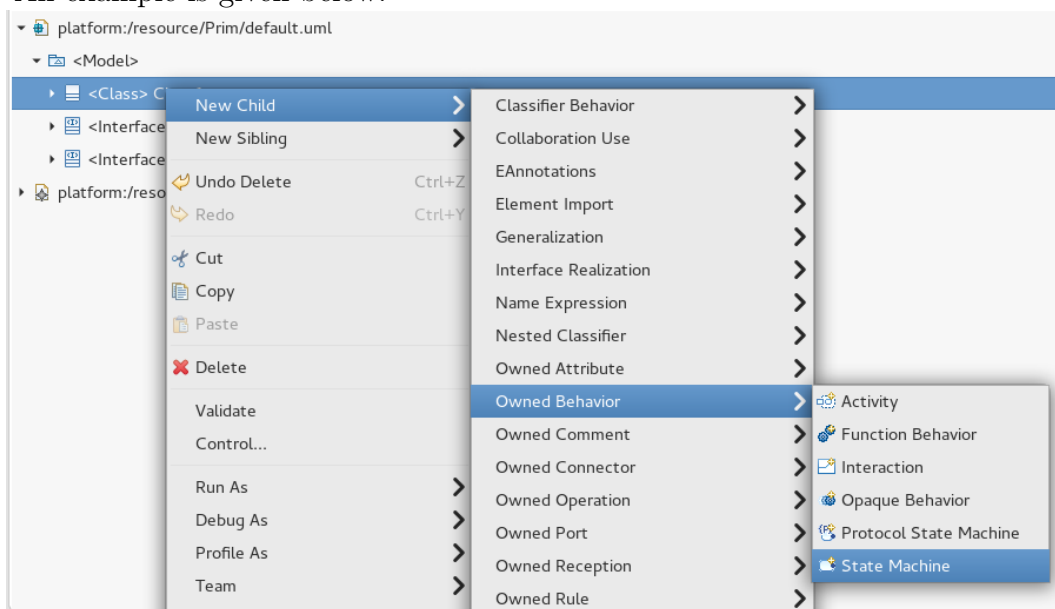
An example of the resulting diagram is given on the figure below.



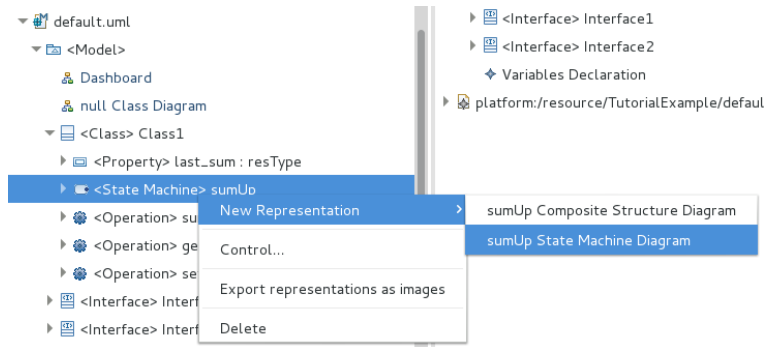
2.11 Create and attach a State Machine to a UML Operation

State Machines define the behaviour of operations of UML classes. In order to specify a State Machine of an operation you need to:

1. Open a .uml file (UML model) in a standard EMF editor and go to the UML Class which operation behaviour you want to specify. Right-click on the class and select **New Child** → **Owned Behavior** → **State Machine**. This will create an instance of the State Machine. You can specify its name in the Properties view. An example is given below.



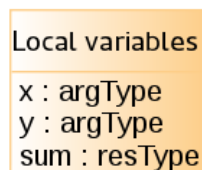
2. Add a region to the State Machine in a similar way. Right-click on the created state machine and select **New Region** → **Region**.
3. Save the model. Now you can create a diagram of your State Machine.
4. Right-click on the created State Machine in Model Explorer and select **New Representation** → **State Machine diagram**. Specify the name of the diagram and hit Ok.



Note! Set/get methods should NOT have State Machines attached. Their implementation is default in the current version of VerCors.

2.12 Edit Local variables of a State Machine

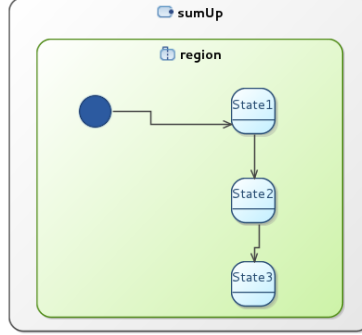
If you want to use local variables on your State Machine, you should declare them in a specific section on a State Machine Diagram called **Local variables**. You should add one box for the local variables declaration to your state machine diagram using a tool **Variable Declaration Area**. Then, you can add local variables to the box using **Variable** tool. To add a variable click on the tool and then click on bottom part of the box for variables. You should set the names and types of variables in the Properties view. Figure below illustrates local variables of our example.



The input parameters of the operation modelled by the state machine should be also included in the list of local variables. Moreover, for each local variable of array type with a name **name** there can be declared a local variable with a name **name_length** which stores by default the length of the array. If the array is a result of a multicast method invocation, its length will be automatically evaluated to the group size. Otherwise, it is automatically evaluated to the size of the array specified in the data type. This is crucial for the code generation and model-checking.

2.13 Edit State Machines

You can use Palette of the State Machine Diagrams to add transitions and states. You can also directly edit labels of states and transitions. To follow our example you can first add states and transitions to your State Machine Diagram as illustrated below.



The behaviour of is specified on state machine transition labels. In order to edit a label you need to click on it once in order to select it. Then, you can click on it second time to edit it. From a state machine you have access to the local methods of the class owning the state machine, to the methods of client interfaces and to the local variables of the state machine. The attributes of the class can be accessed only through get and set methods. The labels text should correspond to the grammar given below.

$$\begin{aligned} \langle \textit{Transition_label} \rangle &::= '[' \langle \textit{Guard_Expr} \rangle ']' \\ &| '/' \langle \textit{Stms} \rangle \\ &| '[' \langle \textit{Guard_Expr} \rangle ']' \\ &| '/' \langle \textit{Stms} \rangle \end{aligned}$$

$$\begin{aligned} \langle \textit{Stms} \rangle &::= \langle \textit{Stm} \rangle ';' \\ &| \langle \textit{Stm} \rangle ';' \langle \textit{Stms} \rangle \\ &| \langle \textit{MCall} \rangle \\ &| \langle \textit{Return} \rangle \end{aligned}$$

$$\langle \textit{Stm} \rangle ::= \langle \textit{Assign} \rangle$$

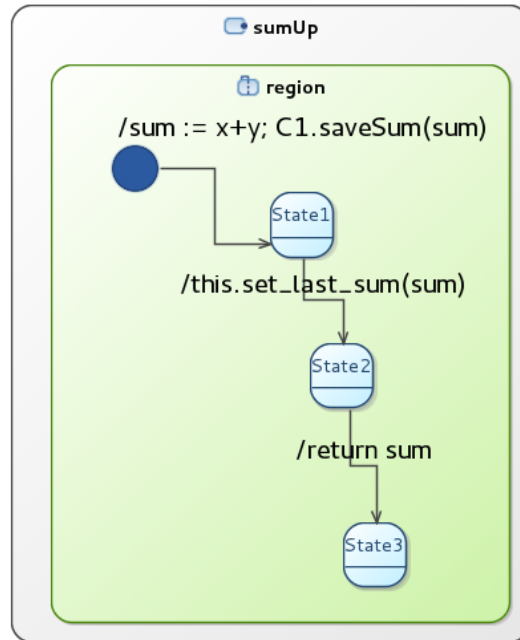
$$\begin{aligned} \langle \textit{Assign} \rangle &::= \langle \textit{Variable} \rangle ':= ' \langle \textit{Expr} \rangle \\ &| \langle \textit{Variable} \rangle ':= ' \langle \textit{MCall} \rangle \end{aligned}$$

$$\begin{aligned} \langle \textit{Expr} \rangle &::= \langle \textit{Variable} \rangle \\ &| \langle \textit{Constant} \rangle \\ &| \langle \textit{Expr} \rangle \langle \textit{Bop} \rangle \langle \textit{Expr} \rangle \\ &| '(' \langle \textit{Expr} \rangle ')' \\ &| \langle \textit{Array_element} \rangle \end{aligned}$$

$$\begin{aligned} \langle \textit{MCall} \rangle &::= \textit{ID} '.' \textit{ID} '(')' \\ &| \textit{ID} '.' \textit{ID} '(' \langle \textit{Args} \rangle ')' \\ &| \textit{'this.'ID '(')}' \\ &| \textit{'this.'ID '(' \langle \textit{Args} \rangle ')' } \end{aligned}$$

$$\begin{aligned}
\langle Arg \rangle &::= \langle Constant \rangle \\
&| \langle Variable \rangle \\
&| \langle Arg \rangle ', ' \langle Arg \rangle \\
\\
\langle Constant \rangle &::= NUM \\
&| \langle Boolean_constant \rangle \\
&| \langle Enum_constant \rangle \\
\\
\langle Bop \rangle &::= '&&' \\
&| '||' \\
&| '+' \\
&| '-' \\
&| '*' \\
&| '/' \\
&| '=' \\
&| '>=' \\
&| '>' \\
&| '<' \\
&| '<=' \\
\\
\langle Boolean_constant \rangle &::= 'true' \\
&| 'false' \\
\\
\langle Enum_constant \rangle &::= ID '. ' ID \\
\\
\langle Guard_Expr \rangle &::= \langle Expr \rangle \\
\\
\langle Return \rangle &::= 'return' \langle Variable \rangle \\
&| 'return' \langle Constant \rangle \\
&| 'return' \\
\\
\langle Array_element \rangle &::= ID '[' \langle Expression \rangle ']'
\end{aligned}$$

The final version of sumUp state machine of our example is given on the figure below.



LIMITATIONS. The current version of the plug-ins dealing with code generation and model-checking do not support:

- Several method calls on the same transition;
- Any kind of expressions except from variables as arguments of method calls.

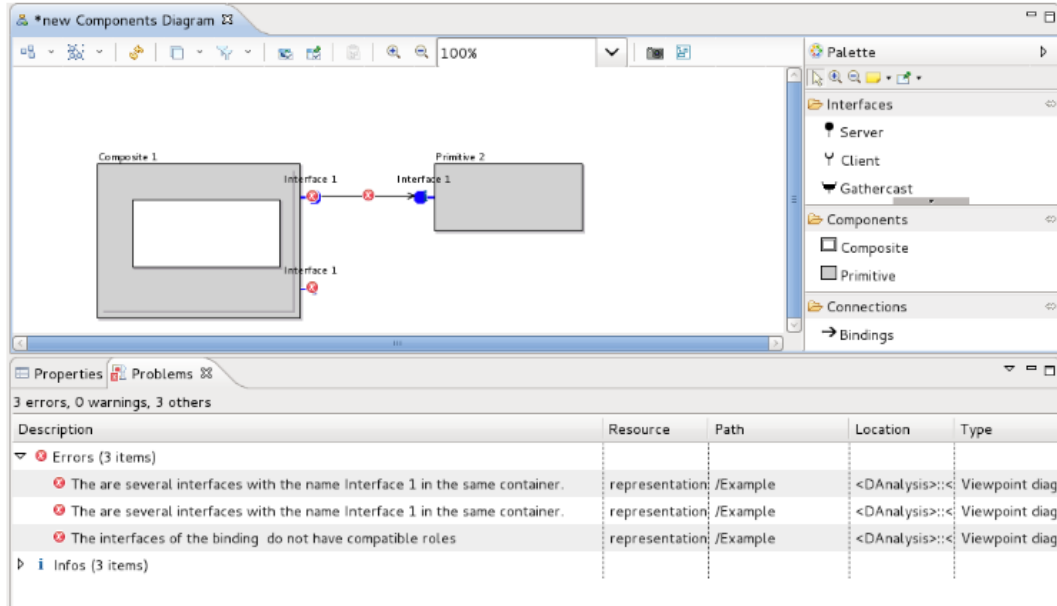
NOTE! For the model-checking, every state machine should have a "return" statement even if it describes a void method.

3 Diagram Validation

Before using the diagrams, and in particular before generating ADL files, it is mandatory to check the structural coherency of the semantics. In order to do this open a VCE Component Diagram and select **Diagram** → **Validate** in the menu.

The elements of the diagram which did not pass the validation should be marked with red signs. You will also see the validation errors/warnings description in the Problems view, and in the Model explorer.

A figure below illustrates an example of a non-valid model. The interfaces of Composite1 did not pass the validation because they have the same names. The binding did not pass the validation because it goes from a server interface to a server interface.



The violation of the following constraints are considered to be an error:

- Bindings do not cross a component border;
- The interfaces connected by a binding have compatible types;
- The interfaces connected by a binding have compatible roles;
- The interfaces connected by a binding have compatible natures;
- All the interfaces in the same container have different names;
- All the components in the same container have different names;
- The violation of the following constraints are considered to be a warning:
- The interfaces connected with a binding should have different containers;
- The violation of the following constraints are considered to have the information level:
- Each GCM interface should have a reference to a UML interface.

3.1 Generating executable GCM/ProActive files

Once a component diagram has been checked valid, it is possible to generate automatically some of the files necessary for building a GCM/ProActive executable application. More precisely the generated files are:

- one ADL file, in XML format respecting the ADL DTD⁴. It represents an application component architecture (components, interfaces, bindings),

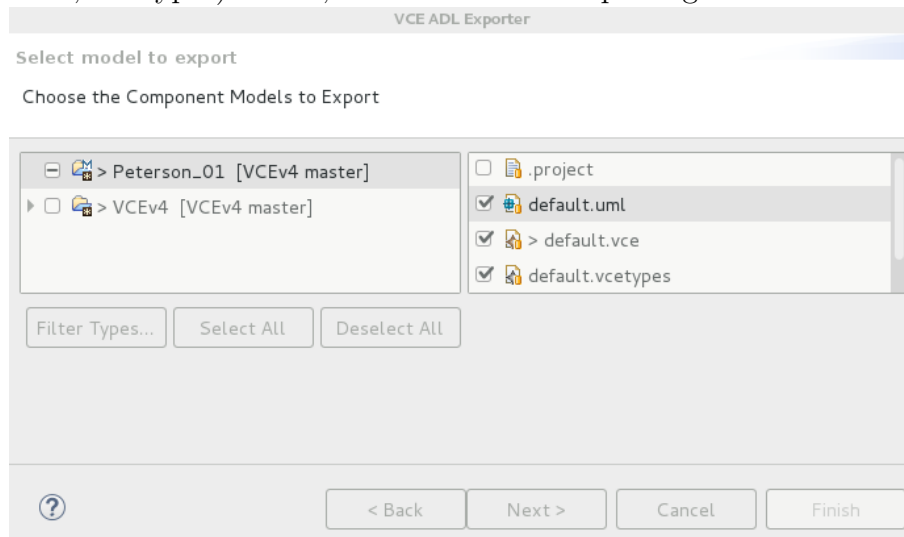
⁴classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd

- one Java interface for each UML interface in the application;
- one Java class per UML class specified on Class diagram;
- one Java interface for every UML class attached to a primitive component with attributes specification (such component becomes an attribute controller in terms of GCM/ProActive and needs an interface to be implemented);
- one Java class per each Enumeration and Record type from the VceTypes model;
- a Java enumeration State used for methods implementation

VerCors partially generates executable code of class operations. For those operations that have State Machines attached it translate State Machines into Java code. It also generates the code of set/get methods to access attributes of the Java classes.

In order to generate code you need to:

1. Right-click on .vce file of your project. Select **Export...**
2. Select **VCE Export** → **ADL Export**. This will open a wizard.
3. Select all the models of your project that should participate in the generation (i.e. .vce, .uml, .vcetypes). Then, hit Next. An example is given below.



4. Give the name to the .fractal file that will be generated and to the package of the generated Java classes. **Note!** those names are mandatory and must be different from each other!
5. Hit Next.

4 Generation of input for CADP

Graphically specified models are translated by VerCors into low-level models encoding the behaviour of a GCM applications. PNetS and pLTS are used to encode the behaviour as described here⁵.

In order to assist model-checking of the specified models, VerCors generates the following files:

- a .fcr file per each pLTS;
- a .exp file per each pNet;
- .svl script that optimizes models and calls script transforming .exp files into .bcg
- run.sh script that calls .svl scripts and flac-and-minimize script on all .fcr files.

To generate a pNet of a composite you need to:

1. Right-click on the representation of the Composite component on VCE Components Diagram and select **Generate pNet** → **Generate pNet**. This will open the wizard where you can step-by-step specify the parameters of the verified system.
2. Select the State Machine of the environment scenario. This state machine should not be attached to any operation and should be stored in the root (Model) of .uml model.
3. Specify the queue size of the root component as well as its sub-components.
4. Specify the communications that you do not want to observe during the model-checking.

LIMITATIONS The current version does not generate the pNets of the interceptors, architectures with gathercast interfaces or external multicast interfaces.

5 Reconfiguration

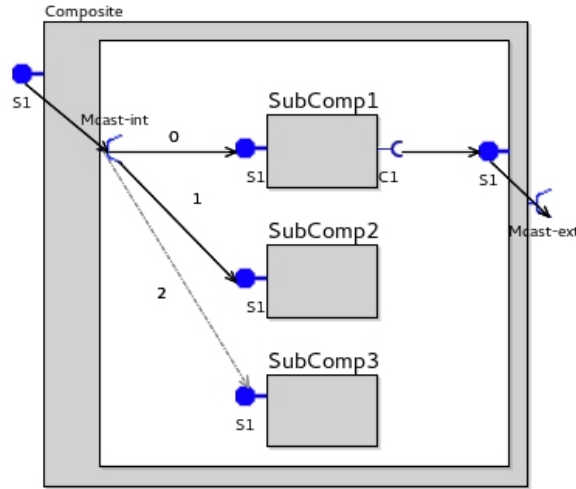
With VerCors, the user can design, generate, and verify a component system with reconfigurable multicast interfaces. A multicast interface (or a *multicast* for short) is a client interface which can send several requests to different targets simultaneously, and then gather the results. Moreover, the group of target interfaces can be reconfigured at run-time, i.e. the bindings going from a multicast interface can be dynamically added and removed.

⁵<https://hal.inria.fr/hal-00761073>

5.1 Graphical design

In order to create a multicast interface, the user should create a singleton interface using the **Interface** tool and then change the **cardinality** of the interface to **collective** and the **role** to **client** in the **Properties view**.

The figure below shows an example of a composite with an external multicast **Mcast-ext** and an internal multicast **Mcast-int**. By editing the labels of the bindings, the user should assign indices to the bindings going from a multicast interface. This is needed in order to refer to them while modelling the reconfiguration of the system. The indices should go from 0 to N where N is the number of bindings going from the interface - 1. In order to analyse application reconfiguration, the user should also model bindings which do not exist when the modelled application is launched but can appear during system execution, e.g. the dashed binding 3 going from **Mcast-int** to **S1** of **SubComp3**. This can be configured by changing the value of the **isActive** property of a binding in the **Properties view**.



The internal interfaces of a composite component can be reconfigured by the non-functional components located in its membrane. In order to specify such kind of reconfiguration the user should:

1. Create an enumeration type which includes the name of the reconfigured interface as an enumeration literal
2. On the transition of a state machine of a component triggering the reconfiguration, the user should use a specific statement corresponding to the reconfiguration instruction. The statement should be of the form `parent.bind(<enum_literal>, <binding_index>)` where `enum_literal` is the enumeration literal corresponding to the reconfigured interface, and `binding_index` is the index of the reconfigured binding. The `unbind` instruction can be defined in a similar way.

For instance, the following instruction corresponds to binding the binding number 2 of our example: `parent.bind(Interfaces.Mcast-int, 2)`. An enumeration type with the name **Interfaces** and a literal **Mcast-int** should be defined by the user. In the future versions such enumerations will be generated automatically.

VerCors generates pNets and Java code for the models with such kind of reconfiguration.