VCE v.3 Tutorial

INRIA Sophia Antipolis Méditerranée

SCALE team

Date:	Nov. 17th, 2014
Authors:	Oleksandra Kulankhina, oleksandra.kulankhina@inria.fr Nassim Jibai, nassim.jibai@inria.fr Eric Madelaine, eric.madelaine@inria.fr Galyna Zholtkevych, galyna.zholtkevych@inria.fr
Version:	0.5
VCE Version	3.1.25

Table des matières

1. Introduction

VerCors is a platform for the specification, analysis, verification and validation of the GCM-based applications architecture and behavior. It contains a set of tools including VCE v.3. VCE v.3 is a graphical designer for the GCM architecture and the Components behavior specification. It is distributed as a set of Eclipse plugins.

This guide explains the basic functionality of VCE v.3.

It provides step by step instructions for the creating a simple example of a Component model using the VCE editors, including all facets of this model: classes and interfaces, types, architecture, and behaviors. It also explains how to validate the static semantics properties of the Component models, and to produce (partial) executable code in GCM/ProActive.

Additional information:

VCE v.3 is based on Obeo Designer technology and is based on the functionality of the standard editors created with Obeo Designer. 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 Obeo Designer documentations. In particular:

- Getting started for End-users: http://docs.obeonetwork.com/obeodesigner/6.1/Getting_Started_User.html
- How to create and manage Modeling Projects: http://docs.obeonetwork.com/obeodesigner/6.1/viewpoint/user/general/Modeling%20Project.html
- How to create and manage diagrams: http://www.obeonetwork.com/group/obeo-designer/page/obeo-designer-reference-document-6-1

1.1. VCE 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**,** and **VTY** models **for** types definition), and 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 use VCE Types for the typing of the arguments and return values.

2. The first example

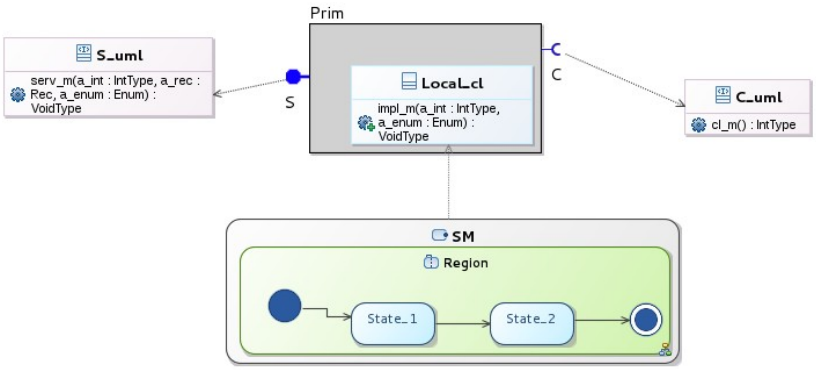
In this section we explain how to:

- create a new VCE Project or import it from an existing source;
- edit GCM Component diagrams;
- specify types;
- create and edit UML classes, interfaces and operations;
- 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;
- create and edit a State Machine and attach it to a UML operation.

All examples used in this tutorial are contained in a tutorial project, available on the VCE web site. We strongly advise the reader to install Obeo, download the Tutorial example archive file, import this project in Obeo using the procedure described below in section 2.3, and follow all steps of the tutorial using the tool.

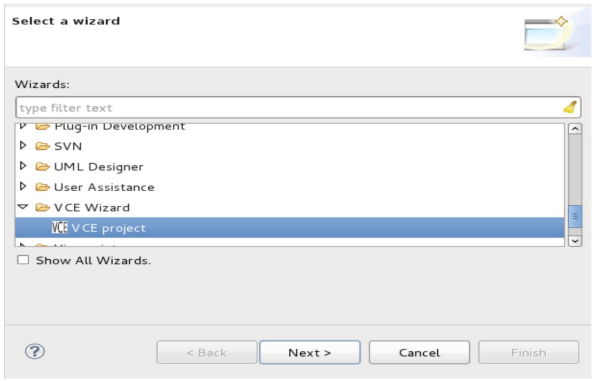
2.1. Example Overview

This section provides an example of a simple VCE project creation illustrated at the figure below. This is a very simple VCE Component diagram containing one primitive component **Prim** with one server and one client GCM interface (**S** and **C**). Each GCM interface is connected to a UML interface (**S_uml** and **C_uml** respectively). **S_uml** has an operation **serv_m** that can be served by the component, **C_uml** has the list of the operations that can be called by the component. A UML class **Local_cl** is attached to the component. It has one method implementing the operation of the server interface (**s_m).** The behavior of the method is illustrated on a UML State Machine **SM**. The method **s_m** takes three arguments: **a_int** of type **IntType**, **a_rec** of type **Rec** and **a_enum** of type **Enum**. These types are specified in the VceTypes model.



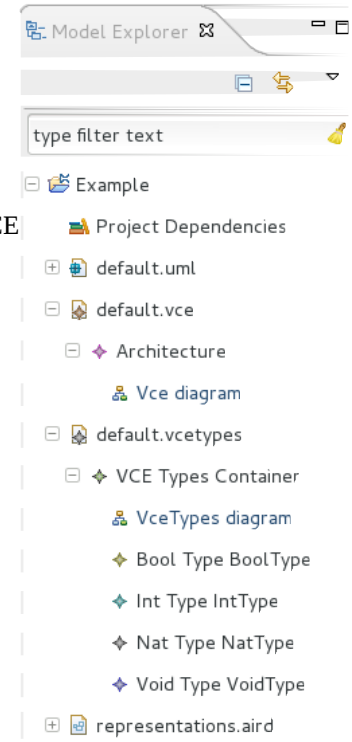
2.2. VCE project creation

- Create a new VCE model: right-click in the Model Explorer and select **“New->Other...”**
- Select “**VCE Wizard->VCE project”** and press on **“Next”**



- Give a name to the project (here ***Example***)
- Press on “**Finish**”

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;
- default.vce – a VCE model;
- VCE diagram – a VCE Component diagram illustraiting the elements of a VCE model specified in default.vce;
- default.vcetypes – a model containing the types specification;
- VceTypes diagram illustrates the types specified in default.vcetypes.

2.3. Using an existing VCE Project

When working in collaborative environments, you will often have to share projects with your collegues. Or you may have to download an existing project, usualy in the form of an archive file, from existing sources. Importing such a project in your VCE workspace works as for any other Eclipse project:

- Import the archive file (e.g. zip file), and save it somewhere on your file system

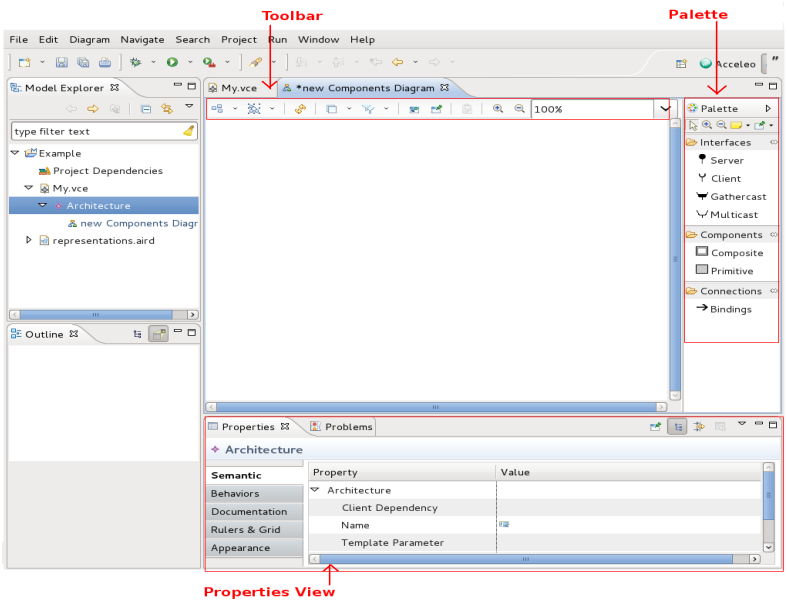
- From the Obeo “**File**”menu, select “**Import**”.
- Select “**General**” / “**Existing project in workspace**” then hit **Next**.
- In the “**Select archive file**” item, **Browse** to find your archive file, hit **Open.**
- Then hit **Finish.**

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 usualy many be more elements in there.
[Hint]: at this point you can rename the imported project...

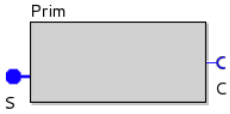
2.4. Building the component Architecture in VCE component diagrams

This section describes how to edit model in VCE components diagram.

- Open VCE components diagram by double-clicking on it. 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-up the tool called “Primitive” and draw a primitive component. Then, add to the component two interfaces using “Server” and “Client” tools. The diagram should look like this:

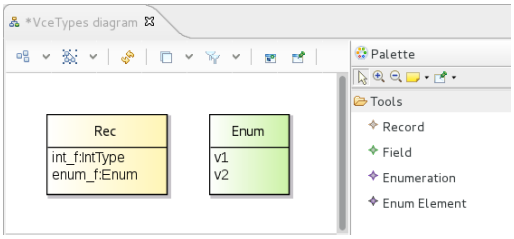


2.5. Types specification

This section describes how to specify the types. VCE has 4 predefined type: **VoidType, BoolType, IntType** and **NatType**. However, one can construct its own types: **Enumerations** and **Records**. Our example has one Enumeration **Enum** with two values: **value1** and **value2**. It also has Record **Rec** with two fields: **int_f** of type IntType and **enum_f** of type Enum**.**

In order to specify the types, you need to:

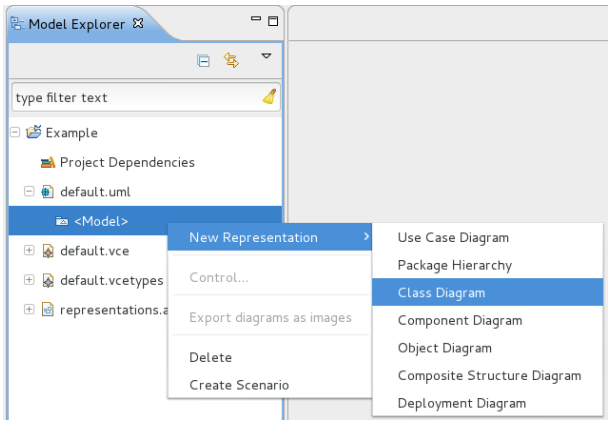
- open VceTypes diagram;
- create an Enumeration using “Enumeration” tool and give it a name using the properties view;
- add two values in the Enumeration using “Enum Element” tool and give them the names in the properties view;
- create a Record using “Record” tool and give it a name “using the properties view;
- add two fields in the Record using the tool “Field” and specify their types and names in the properties view;
- save the diagram.



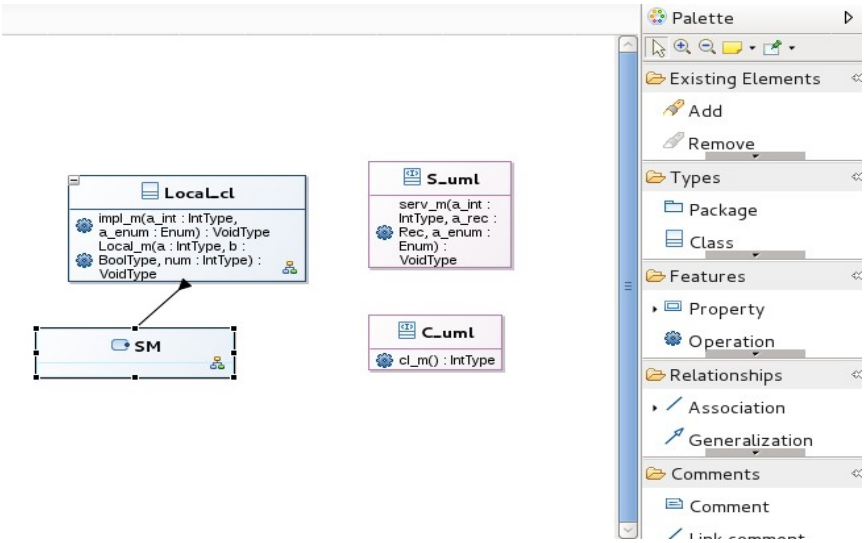
2.6. UML Classes, Interfaces, and Operations specification

The UML Classes, Interfaces and Operations that will be used to specifiy the GCM interfaces and the primitive component implementation class mustbe described using a UML Class diagram. In order to create it, right-click on “**<Model>**” in the Model explorer and select “**New Representation -> Class diagram**”. The new diagram will be created and opened automatically.

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



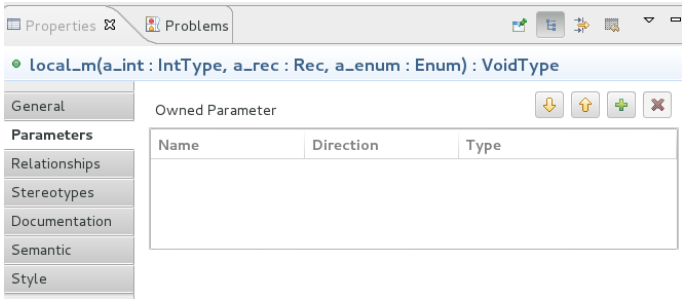
The Class diagram of our example is given below.



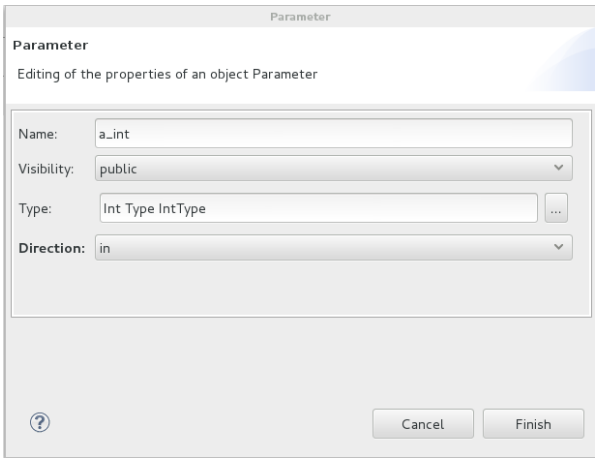
It has two UML Interfaces: **S_uml** and **C_uml**, and one UML Class **Local_cl**. In order to create them, **"Interface"** (resp. **"Class"**) tools are used.

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 Class\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:

- go to the Parameters tab of the Operation Properties view;



- click on the "**green plus**" button in the top-right corner of the Properties view; this will create a parameter;
- you should see a window for the specification of the name, type and direction of the parameter. You can specify here either an argument of an operation, or its return type. Enter its **name** if you are specifying an argument; select the **type** among the ones declared before on VceTypes diagram; select the **direction** "**in**" for the arguments or "**return**" for the return type. The example of **a_int** argument of **local_m** operation is given below.



- finally, save the Class diagram;

You have the possibility to specify an array as a type of a parameter or return type. In order to do this, you need to:

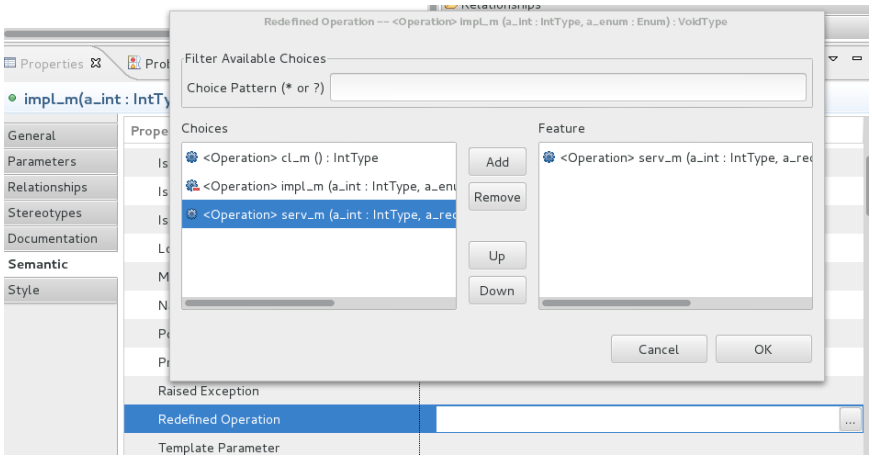
- go to the Semantic tab of the Operation Properties view;
- find the parameter that you want to specify as an Array;
- go to property called Upper;

  instead of “1” type the size of your array (if you do not know the size of the array, put -1, instead of this number “***”** will appear, that means that the array has unlimited size).

2.7. Connect a Class Operation to an Interface Operation

Some methods of the UML Classes implement the ones declared on the UML Interfaces. In our example, **impl_m** of **Local_cl** implements (defines) **serv_m** of **S_uml**. order to establish this relationship you need to:

- go to the Properties view of **impl_m** operation;
- go to the **Semantic** tab;
- for the **Redefined Operation** parameter select **serv_m;**
- save the diagram;

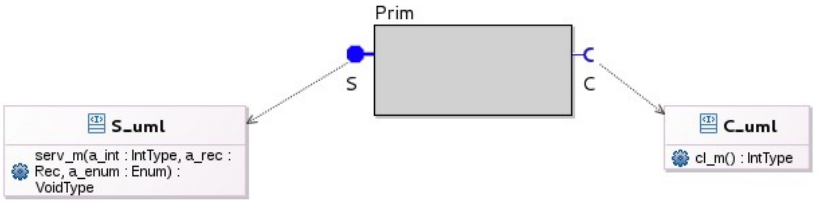


2.8. Attach a UML Interface to a GCM Interface

In our example, a UML Interface **S_uml** is attached to a GCM Interface **S** and **C_uml** is attached to **C.** In order to attach a UML Interface to a GCM Interface you need to:

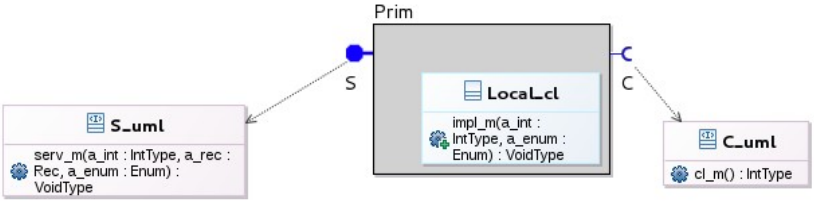
- open the VCE Components Diagram (Vce diagram in our example);
- right-click on the GCM interface to which you want to attach a UML Interface (Interface S in our example);
- select "**Attach UML Interface**" option
- select the required UML interface from the given list (S_uml in our example)
- repeat the same actions for the interface C.

If you specify everything correctly, you should get the following diagram for our example:



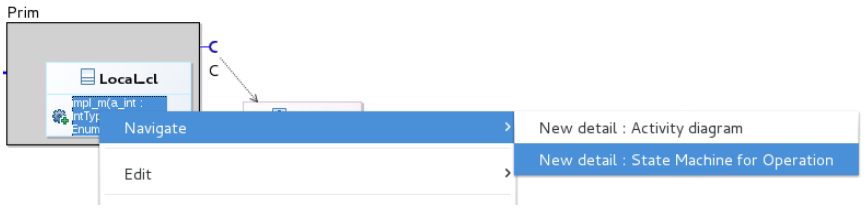
2.9. Attach a UML Class to a GCM Component

The UML Classes illustrate the lists of the operations that can be processed by the GCM Primitive Components. In our example, **Local_cl** contains the methods of a component **Prim**. In order to specify this relation, 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 below:

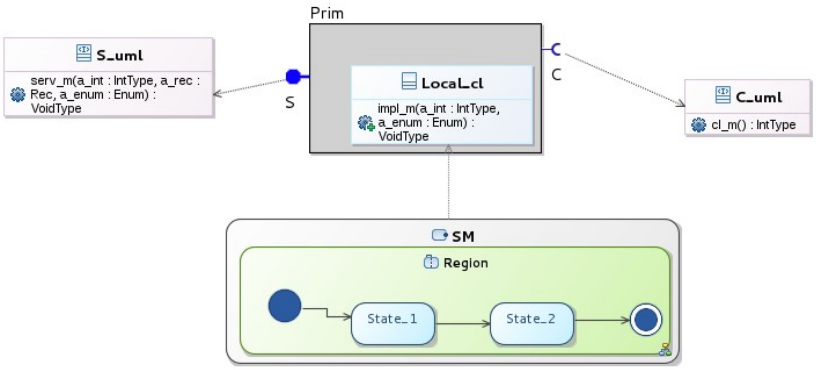


2.10. Create and attach a State Machine to a UML Operation

In order to create and attach a State Machine that specifies the behavior of an operation, right-click on a UML operation of a UML Class and select "**Navigate -> New detail: State Machine diagram for operatio**n". You can do this either on a UML class representation in a VCE Components Diagram, or in a UML Class Diagram.



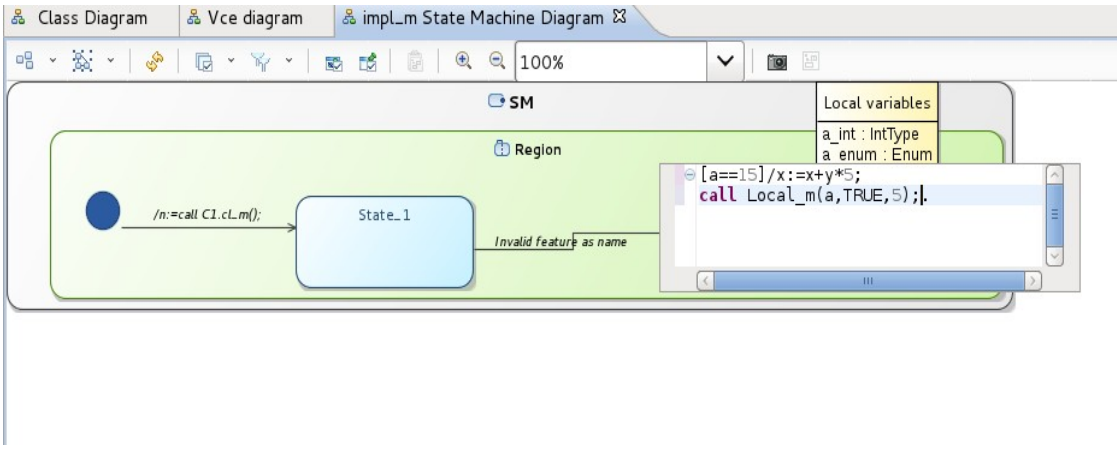
The new State Machine will be created and its diagram will be opened automatically. You can edit the diagram using the tools on the Palette panel. You should save the diagram. After this, you will be able to see the State Machine on the VCE Components Diagram (Vce diagram in our example).



2.11. Edit the State Machine transitions labels

VCE is using an ***Xtext*** embedded editor for the construction of the State Machine labels. In order to open the editor, go to the State Machine diagram, right-click on a transition and select "**OpenEmbeddedXtextEditor** ".

One great benefit of Xtext is that it knows about the context of your diagram, and in particular of the client interfaces that are available from a specific state-machine, and the method calls (services) accessible through these interfaces. This allows you to use auto-complete (Ctrl+Space) when building the labels.



Here are examples of labels taken from the Tutorial project example:

/n:=call C1.cl_m(); x:=11;.

Service **cl_m** from client interface **C1** is called (with no argument), and its return value is assigned to the local variable **n**; then variable **x** is assigned.

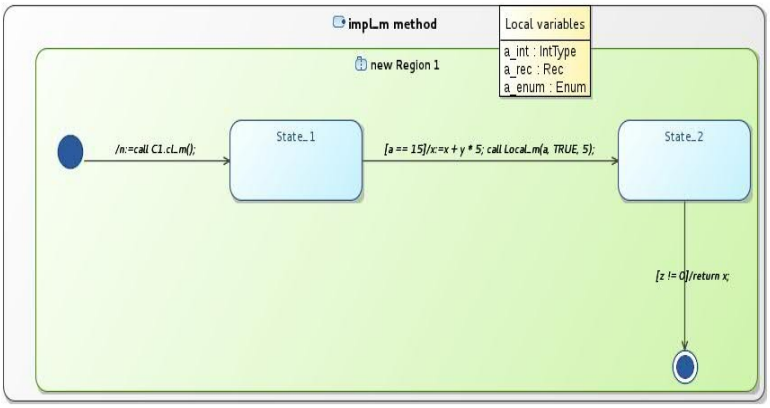
[a == 15]/ x:=x+(y*5); call Local_m(a, TRUE, 5);.

This transition has a guard, and will only be available if variable a is equal to 15. In this case a new value for variable x is computed, than the local methos Local_m is called.

[z != 0] /return x;.

If variable **z** is not null, the current method call with return, with the value of variable **x**. Note that static semantic conditions may check that the next state is a final state of the state-machine.

Here is the complete example of the state machine following current examples for constructing transition labels.



The full grammar for the construction of the labels is given by the following BNF description, in which:

- Non terminals are **Capitalized**, terminals are **UPPER-CASE**
- The structure of method calls is as follow:

- either a qualified name **Itf.m**, in which **Itf** is the name of a client interface of the current component, and **m** the name of a method declared in Itf
- or **m** the name of a local method declared in the inner class of the current component.

```
Transition_label = “[“ Guard_Expr “]” “/” Stms  “.” |  “[“ Guard_Expr “]” “.” | “/” Stms  “.”
Stms = Stm “;” | Stm “;” Stms
Stm = MCall | Assign | Return
Assign = Variable := Expr | Variable := MCall
Expr =  Variable | Constant |  Expr Bop Expr | “(“Expr”)”
Mcall = ”call”  ID”.”ID “( )” |
          ”call”  ID”.”ID “( “ Args” )” |
          ”call”  ID “( )” |
          ”call”   ID “( “ Args” )”
Args =  Constant | Variable | Constant “,” Args | Variable “,” Args
Constant = NUM | Boolean_constant
Bop =  &&  |  ||  | + | - | * | / | == | >= | <=|!=
Boolean_constant = “true”|”false”
Guard_Expr = Expr
Return = “return” Variable | “return” Constant |  “return”
```

There are a number of technical difficulties related to the current versions of Obeo designer and Xtext, that may cause bugs or unwanted behavior in some specific circumstances. This imposes limitations to the way you can safely manipulate the state-machines and their labels, that we detail below. Most of these limitations will be solved in the next version of VCE.

- **Do not** :

  -delete a state machine from the model;

  -delete transitions or transition labels once you started construct transition labels;

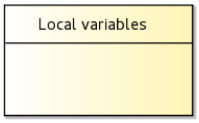
  -delete states, because with states the corresponding transition will be deleted. You may delete state only after that you had changed the source or destination of a transition.

  -change names of methods and their signature in UML Class Diagram if you have already used them in the constructing the State Machine;

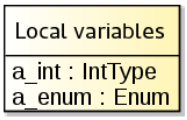
  -attach UML Interface and UML Class for GCM Interface and Component respectively;
- **Do not forget** to save the model after that you had created and defined a VCE Diagram, a UML Class Diagram and a State Machine Diagram.
- Sometimes you have the following situation: when you construct a transition label, mouse cursor flies to the file that consists your text written on the transition (it happens when you put the cursor on a method call. The reason may be that the file transition.vcex, that you can find in the root of the project (file that consists all the text that you had written at transition labels), is not correct. In this case make sure that you followed before all the advices given above. If it is correct, you have to relaunch Obeo.

2.12. Edit the Local variables of a State Machine

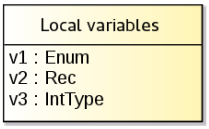
If you want to use some local variables on your State Machine labels, you should declare them in a specific section on a State Machine Diagram called “Local variables”.



The local variables corresponding to the arguments of the Operation to which the State Machine is attached, will be generated automatically when the State Machine is created. For example, for **impl_m (a_int : IntType, e_enum : Enum)** operation of a Class **Local_cl**, the following local variables will be generated:

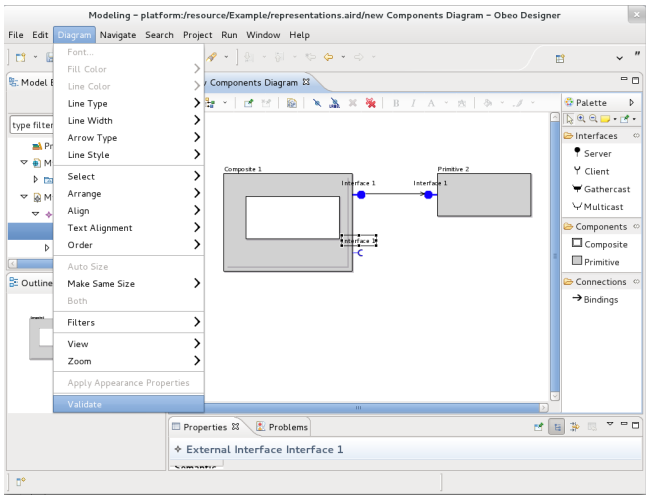


In order to add a new variable, click on the tool “**Variable**” (Palette/Variables) and then click on the box “Local variables” illustrated above. You can change the name and the type of a variable using its Properties view. For the typing you can use any type defined in the .vcetypes model. An example of three variables declaration is given below.



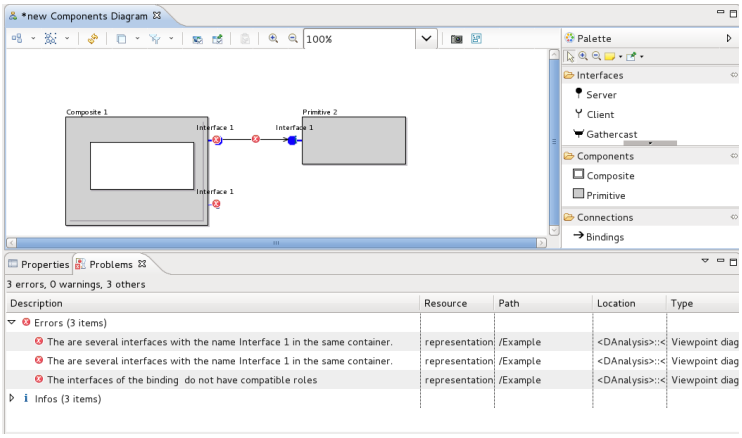
3. Diagram Validation

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 Components 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**" tab, and in the Model explorer.

In our example we got the following result:



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.

A set of constraints is checked. The violation of a constraint can be presented as INFORMATION, WARNING or an ERROR.

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;
- Each GCM interface should have a reference to a UML interface.
- Each GCM Primitive component should have a Referenced class

The violation of the following constraints are considered to be a warning:

- The interfaces connected with a binding should have different containers;

4. Generating GCM/ProActive files

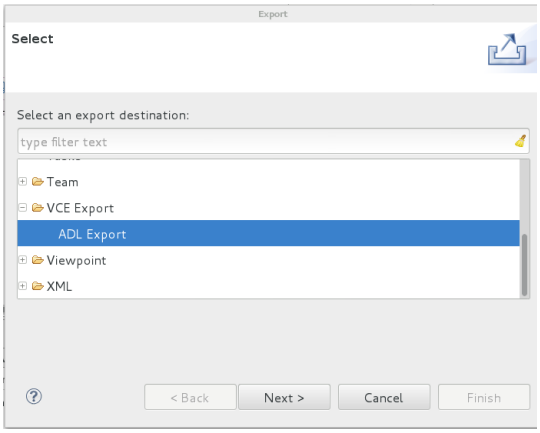
Once a component diagram has been checked valid, it is possible to generate automatically some of the files necessary for building a GCM/ProActive executable application. More precisely the generated files are:

- one ADL file, in XML format respecting the ADL DTD "classpath://org/objectweb/proactive/core/component/adl/xml/proactive.dtd". It represents the application component architecture (components, interfaces, bindings), and the link with the interface and content Java classes,
- one Java interface file for each UML interface in the application
- one Java class for each UML class in the application
- one Java enumeration for each user-defined enumeration type
- one Java class for each user-defined record type

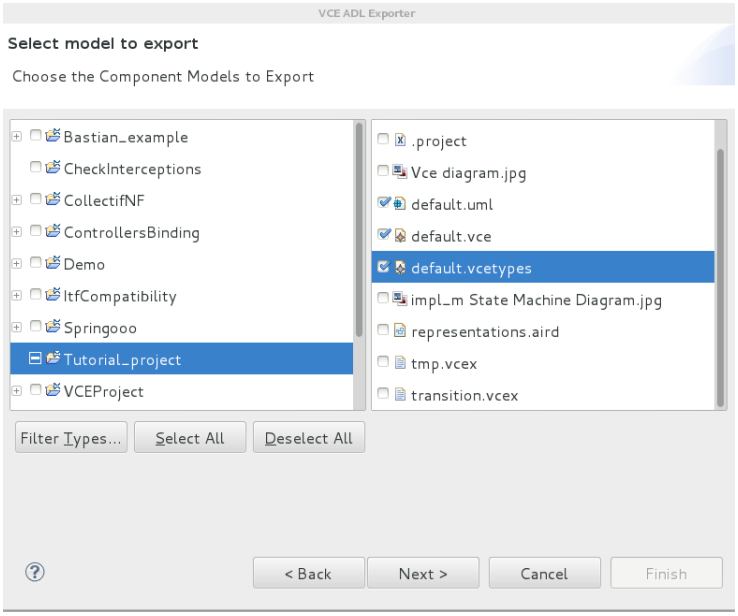
We strongly advise you not to attach the same UML class to more than one primitive component, because only one Java class is generated for each UML class. Hence, if the same class if attached to two different primitive components, you will have to implement their business logic in the same way, and they will have the same collection of generated client interfaces.

In order to launch the generation, you should do the following:

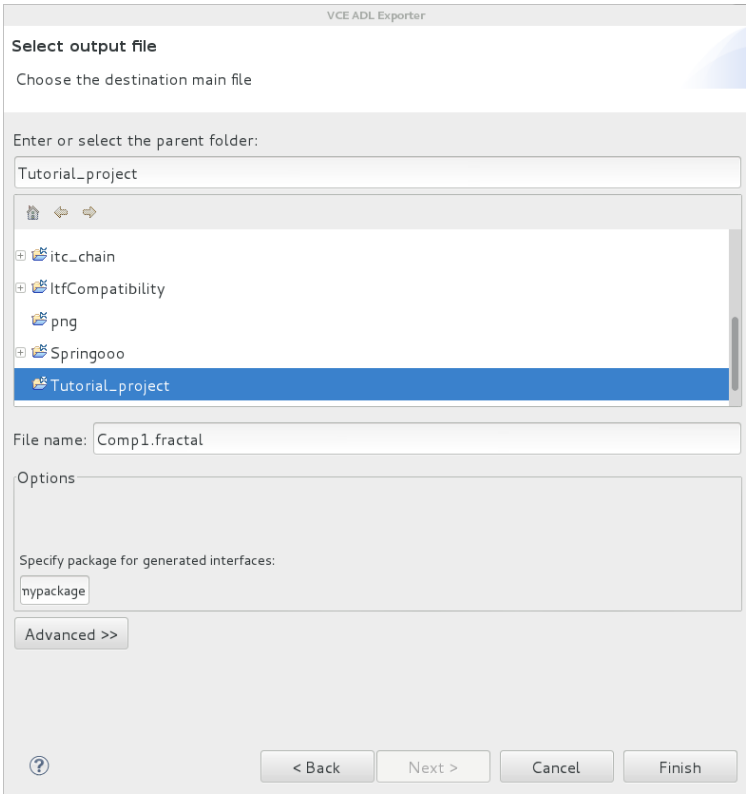
Right-click on your **.vce** file and select "***Export***" option. This will open a wizard. Select "***VCE Export -> ADL Export***" and hit "***Next***".



Select the **.vce**, **.uml** and **.vcetypes** files of your project and hit "***Next***".



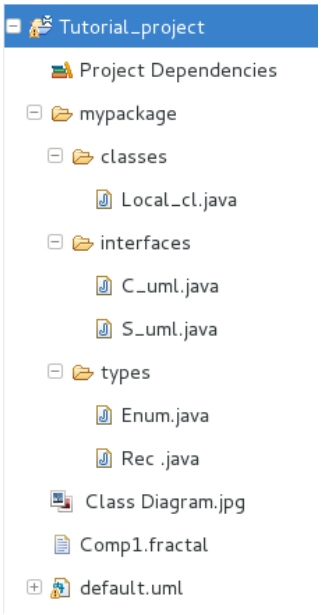
Type the file name ***<Name_of_the_root_component>.fractal.*** In our example it is "***Comp1.fractal***". Type the name of a package. It will be the package for the generated Java classes and interfaces. In our example the name is "***mypackage***" Hit "***Finish***". Please, note, that the ADL generation will trigger validation of the .vce model. If the model is not valid, the ADL description will not be generated.



The generator will create:

- a file ***<Name_of_the_root_component>.fractal*** with ADL description of your architecture;

- **<package_name>/interfaces** directory with Java interfaces;
- **<package_name>/classes** directory with Java classes;
- **<package_name>/types** directory with Java classes and enumerations for the types;



One of the directories might be not created if its containment is not generated (for example, if you do not have any user-defined types).

If you do not see the generated packages in Eclipse, refresh your project: right-click on your project and select "***Refresh***" option.

A Java class is generated for each UML class. It has the following list of operations:

- one method per each UML operation declared in the UML class;
- the methods inherited from ***org.objectweb.fractal.api.control.BindingController*** if the class is attached to a primitive component with client interfaces;
- the methods inherited from ***org.objectweb.proactive.core.component.interception.Interceptor*** if the class is attached to an interceptor.

A Java class generated from a UML class implements the following Interfaces:

- ***java.io.Serializable***
- ***org.objectweb.fractal.api.control.BindingController*** if the class is attached to a primitive component with client interfaces;
- ***org.objectweb.proactive.core.component.interception.Interceptor*** if the class is attached to an interceptor.
- all the server interfaces of the primitive component to which the class is attached.

Please, note, that if a UML class is attached to a primitive component with server interfaces, we do not generate the operations of the server interfaces in the Java class, if their declaration is not duplicated in the UML class description. In our example, ***mypackage.classes.Local_cl*** will have operations ***impl_(Integer a_int, Rec a_rec, Enum a_enum)*** and ***Local_m(Integer a, Boolean b, Integer num)*** . It will implement ***mypackage.interfaces.S_uml*** interface, but it will not have ***ser_m(...)*** method, because it was not declared in the corresponding UML class. We also do not generate attribute of a class even is they are specified on a UML diagram.

In some cases the generated files cannot be compiled when they are imported in a GCM/Proactive project. Here are the know reasons and solutions:

- a Java class implements one of the server interface, but it does not have all the implemented methods. It happens, because those methods were not declared in the corresponding UML class. If you want to fix this issue in the Java code, you need to manually add the implemented methods. Normally, Eclipse should help you to do this. If you want to avoid this issue at the project design stage, you should duplicate all the methods of the serer interfaces, that are attached to a primitive component, in the class attached to the component.
- one of the packages was not generated (.types, .classes, .interfaces). In order to fix this, you have add this package manually.
- A Java class or an interface has two methods with a signature that cannot be compiled. We do not do any type-check for the signatures of the methods in the UML Designer. Hence, a user has to take care of this either at the design or at the implementation stage.

We are currently working on providing the user possibility to set the contingency of the GCM interfaces. However, in the current version we automatically set the contingency of some interfaces during the ADL generation. This feature is currently under construction, hence, sometimes you will have to manually change the value of contingency in the generated ADL files.

5. Examples of VCE projects

The examples of simple diagrams created with VCE v.3 are given below. The projects containing the examples can be found on the VCE download web page:

http://www-sop.inria.fr/oasis/Vercors/software/VCE-v3/installation.html

in the archive files named:

Tutorial Project

All the examples from the previous pages of this document.

Multicast-Gathercast Example

section 5.3 below

BFT Use Case

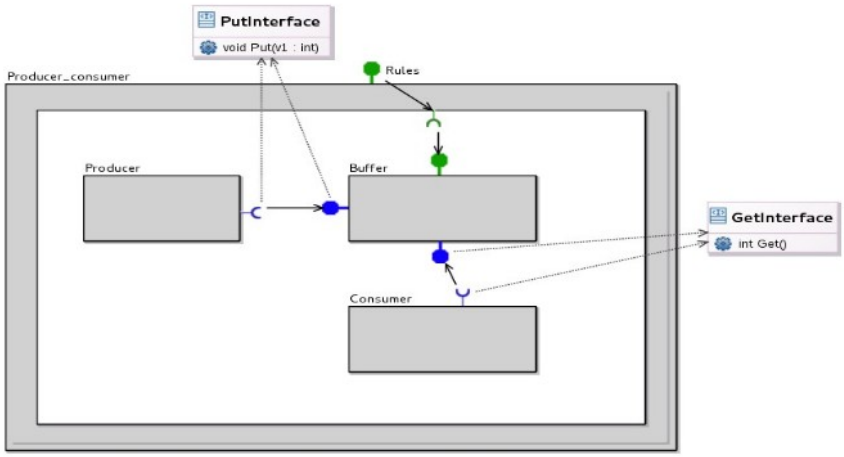
section 5.5 below

You will find also below some additional examples of architecture diagrams, illustrating more features of GCM.

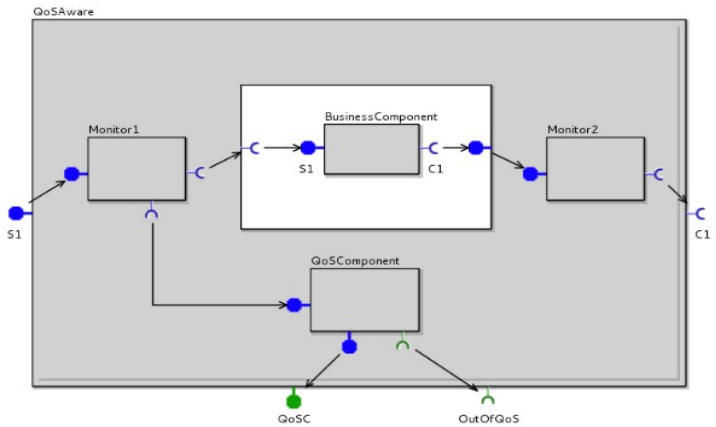
5.1. Composite example

This example of architecture diagram illustrate on a very simple case the standard way of building a composite diagram, with sub-components inside its content, bindings between these sub-components. Note that on this example, the server/client interfaces on each binding have been associated to the same UML interface, so they are compatible by construction.

This composite is a closed application, that runs autonomously. The only possible interaction with the environment is through the “rules” non-functional interface, where the user is supposed to be able to change the configuration of the buffer component.



5.2. Non-functional components example



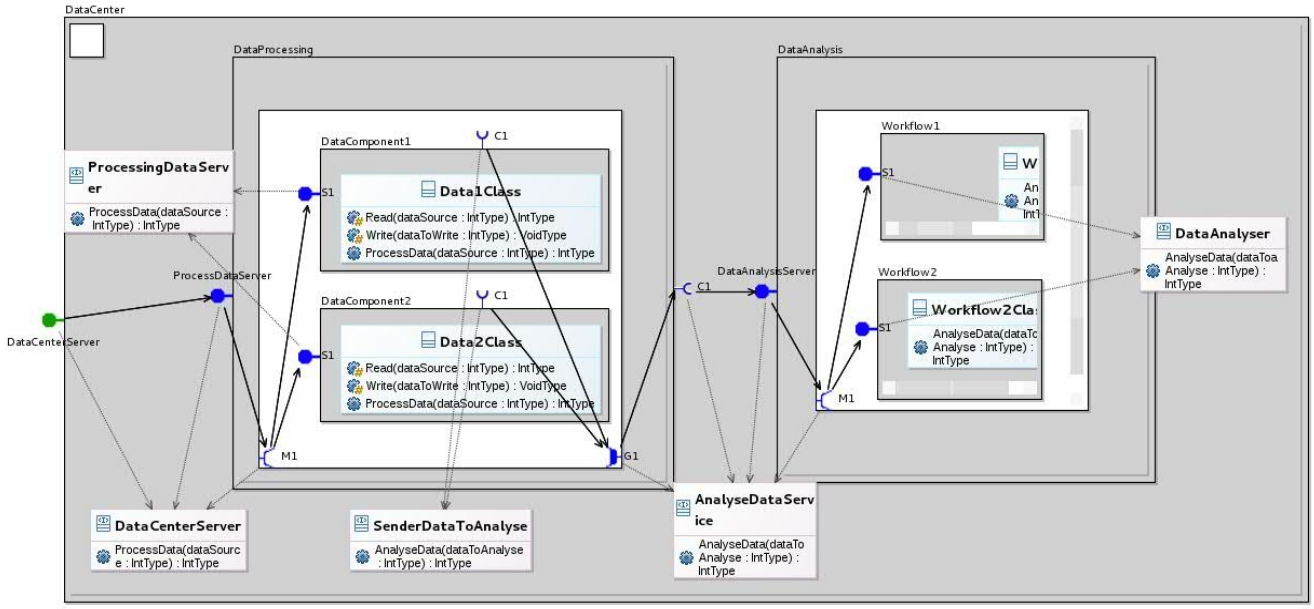
This is an example demonstrating the use of non-functional components in the membrane, for monitoring the quality

of service (e.g. response time) of some business service.

– Monitor1 and Monitor2 are Interceptor components, placed on the data-flow of functional requests to the business component, both on its service interface S1, and on one client interface C1.

– Information from interceptors is used by some non-functional component (here QOSComponent), that may be able typicaly to detect QoS violations and report them to some external observer on the non-functional interface OutOfQoS.

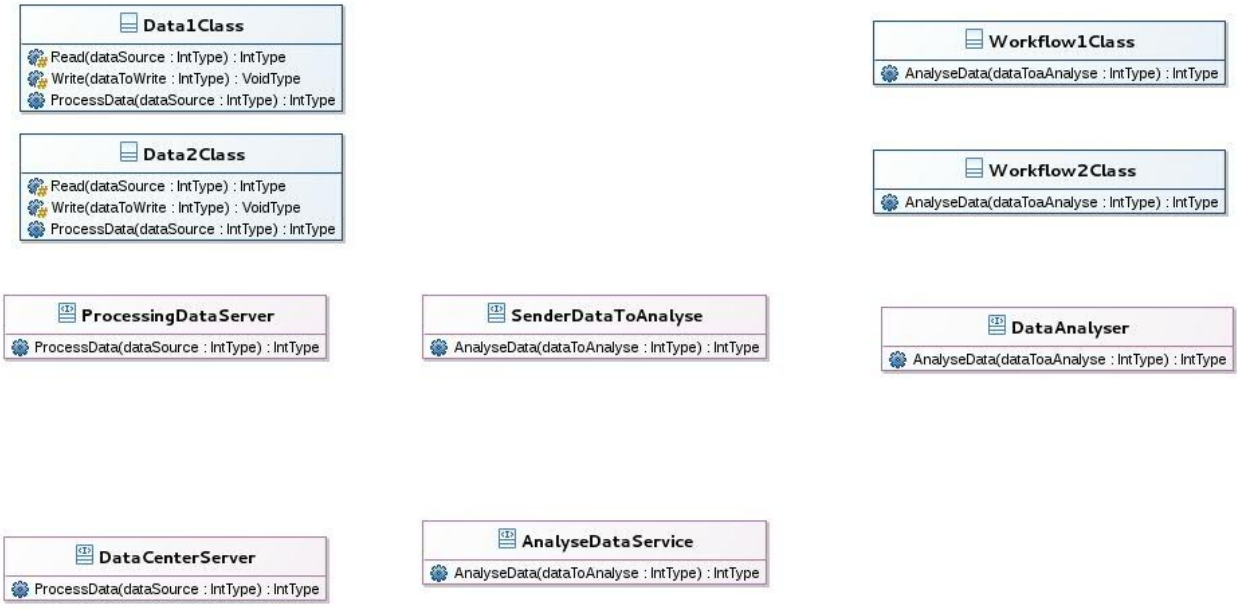
5.3. Multicast / Gathercast example



This is a typical High performance component architecture, where the work is dispatched on groups of components (e.g. {DataComponent 1,2}) through a multicast interface M1.

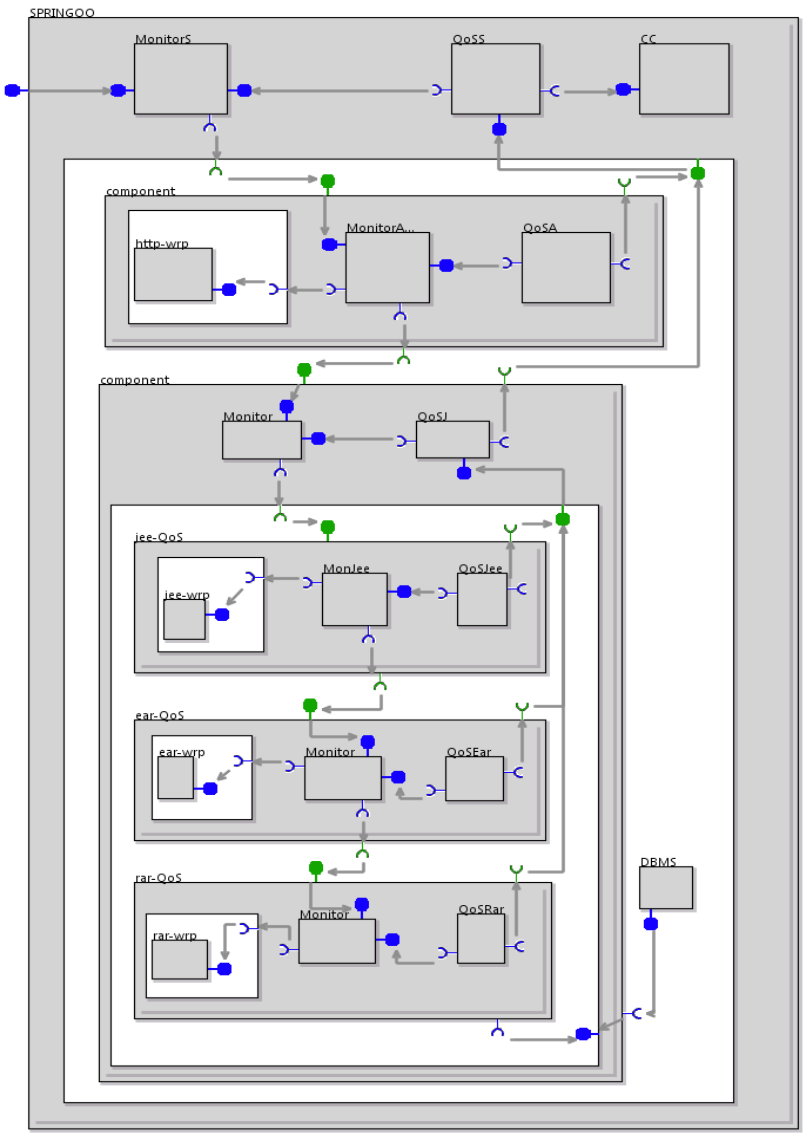
In turn the members of this group have a client interface C1 on which they can emit requests “AnalyseData”. These requests are synchronised and grouped by the Gathercast interface G1, and further routed to another group of processing components.

- Class diagram



5.4. Three tiers application example

This bigger exemple is taken from the french collaborative FUI project “OpenCloudware”. It represents the high-level architecture of a classical 3-tiers web application named SPRINGOO, in which an Apache server receives a flow of requests from users, that are pass to a (set f) JEE servers. These are in charge of processing the requests, depending on their type, and eventualy using data from an SQL database. All components in the architecture are monitored, and the monitoring information is used to control some QoS contracts, and eventualy to adapt the computing resources, especialy the JEE servers, to the users demand.



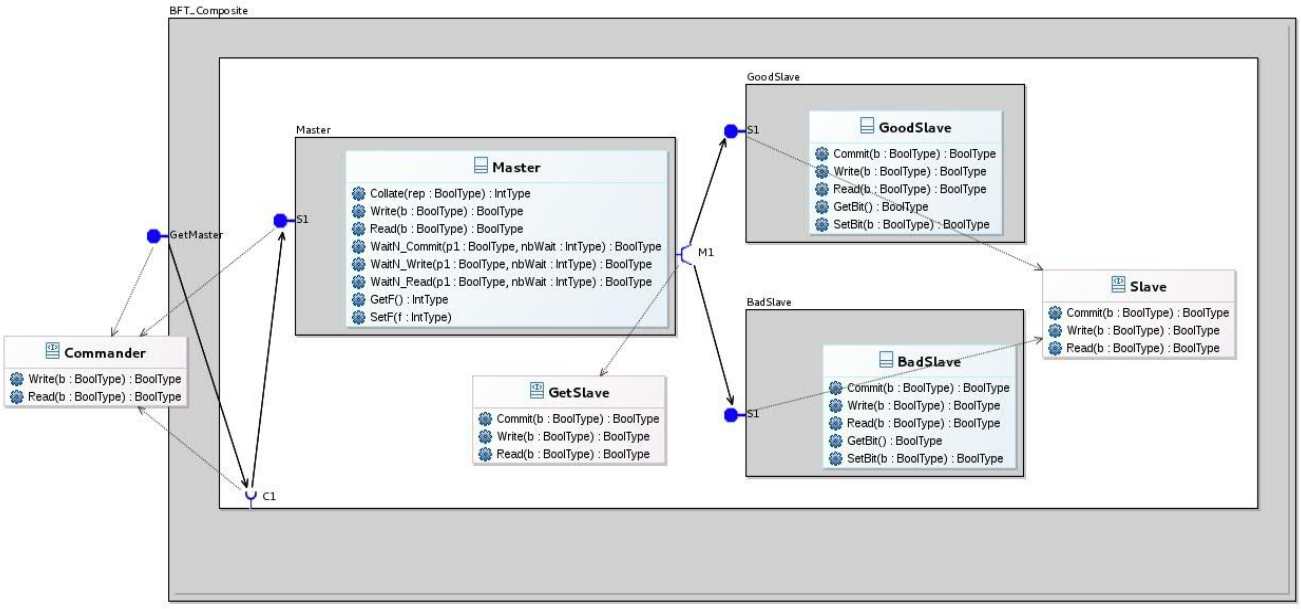
- ## 5.5. Bizantine Fault Tolerant Protocole example

This example was created following the paper *Verifying Safety of Fault-Tolerant DistributedComponents* .

The purpose of that system is next: the system get as input the command of writing or reading. The Component **Master** get this command through GCM interfaces and, in turn, sends the corresponding command to components called slaves (**GoodSlave** and **BadSlave**). The sending to slaves is done by Multicast GCM interface (**M1**). **Master** is waiting for the reply from slaves. They send replies as **BoolType** variables. **GoodSlave** is writing or reading the value and sends back the read or written value and the true reply. **BadSlave** have different implementation. It may behave in any way and may reply with any true or false **BoolType** variable.

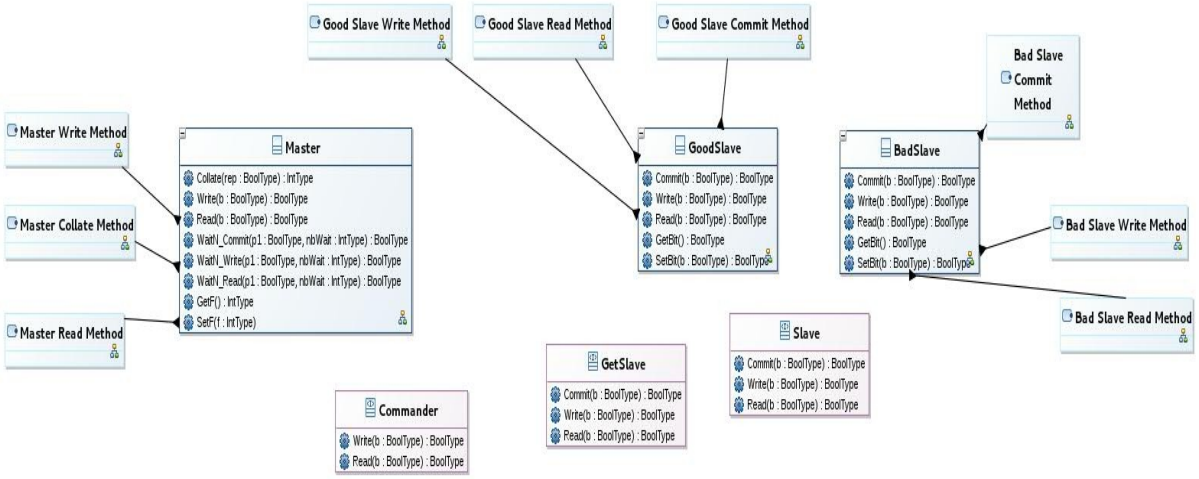
Multicast obtains replies from slaves as an array of variables of **BoolType**. **Master** is counting the number of false and true replies by the method **Collate** and returns true reply if good replies are greater than bad ones. Similarly, it is done for false reply from **Master** component.

- VCE diagram:

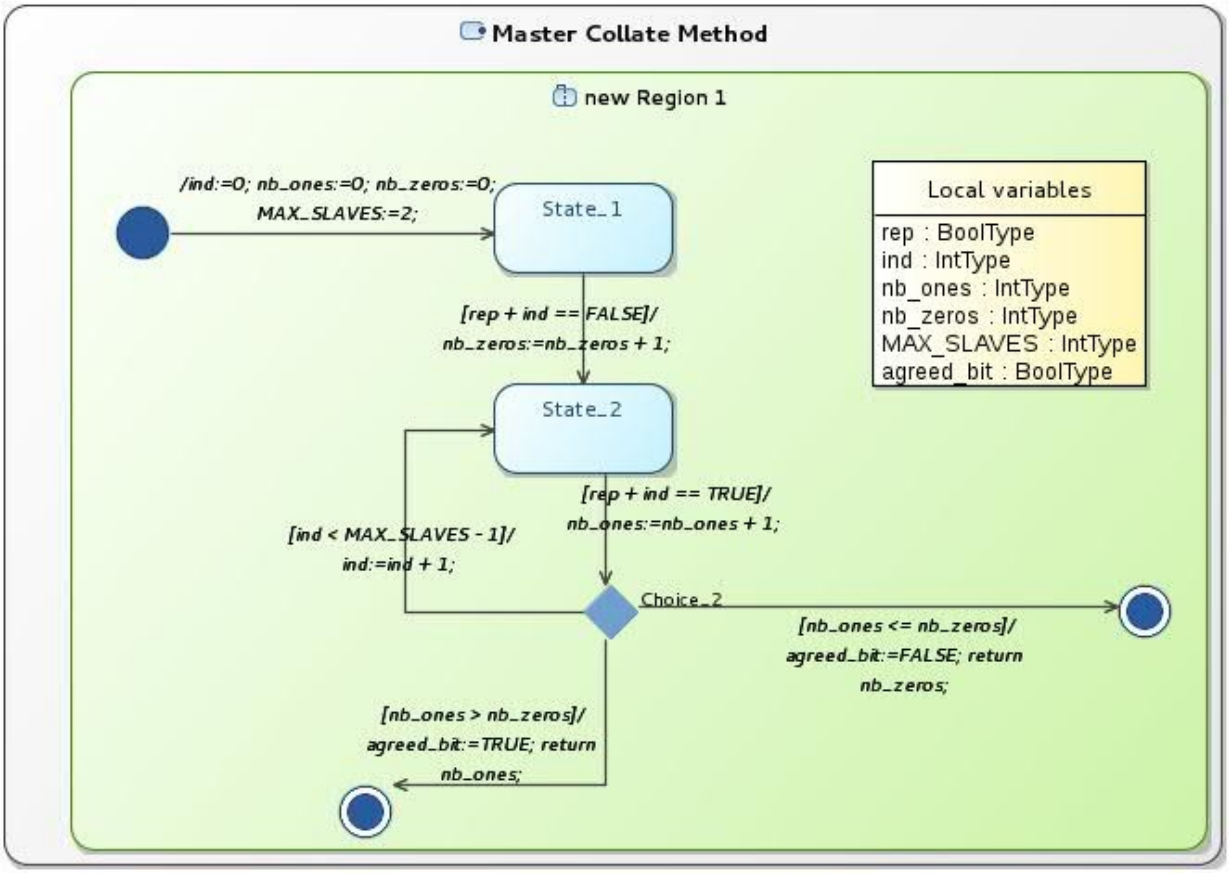


-

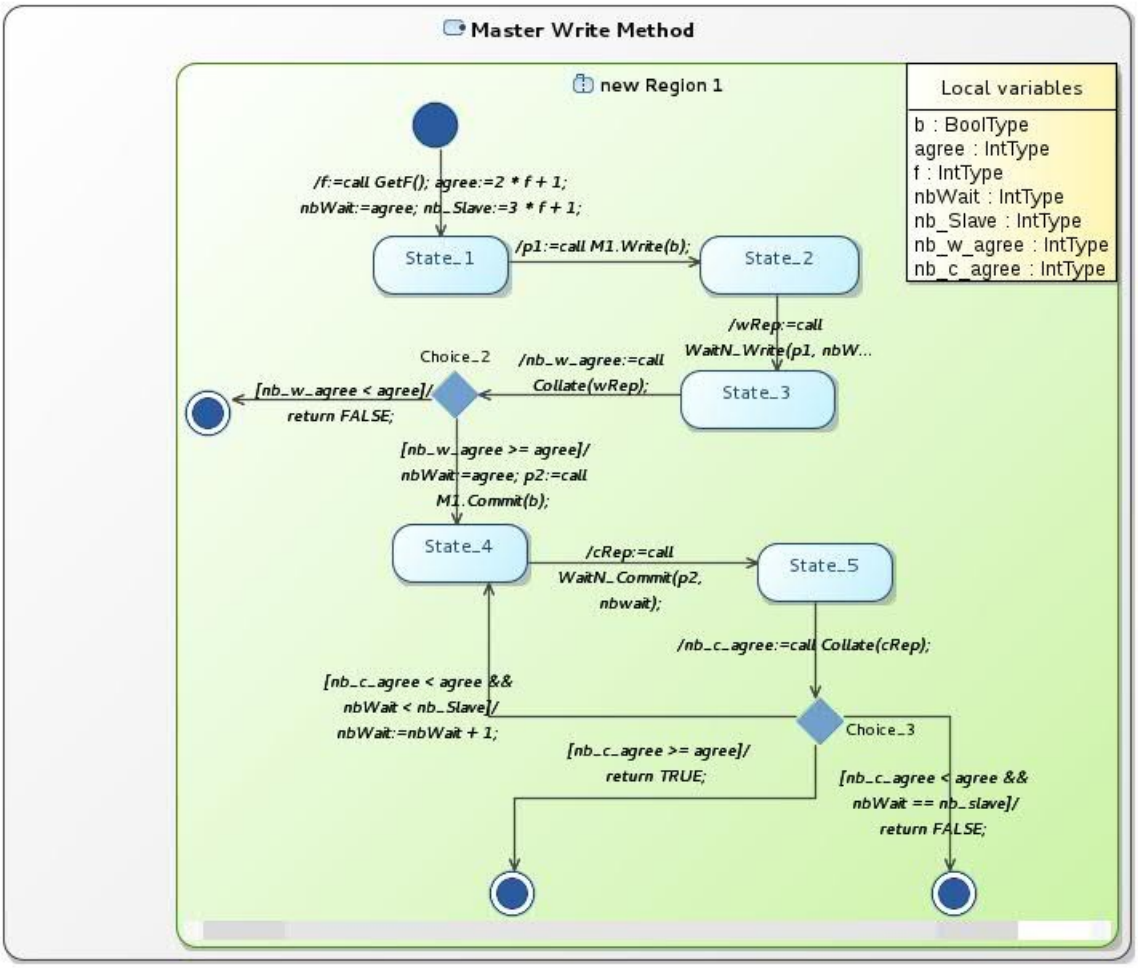
- Class Diagram



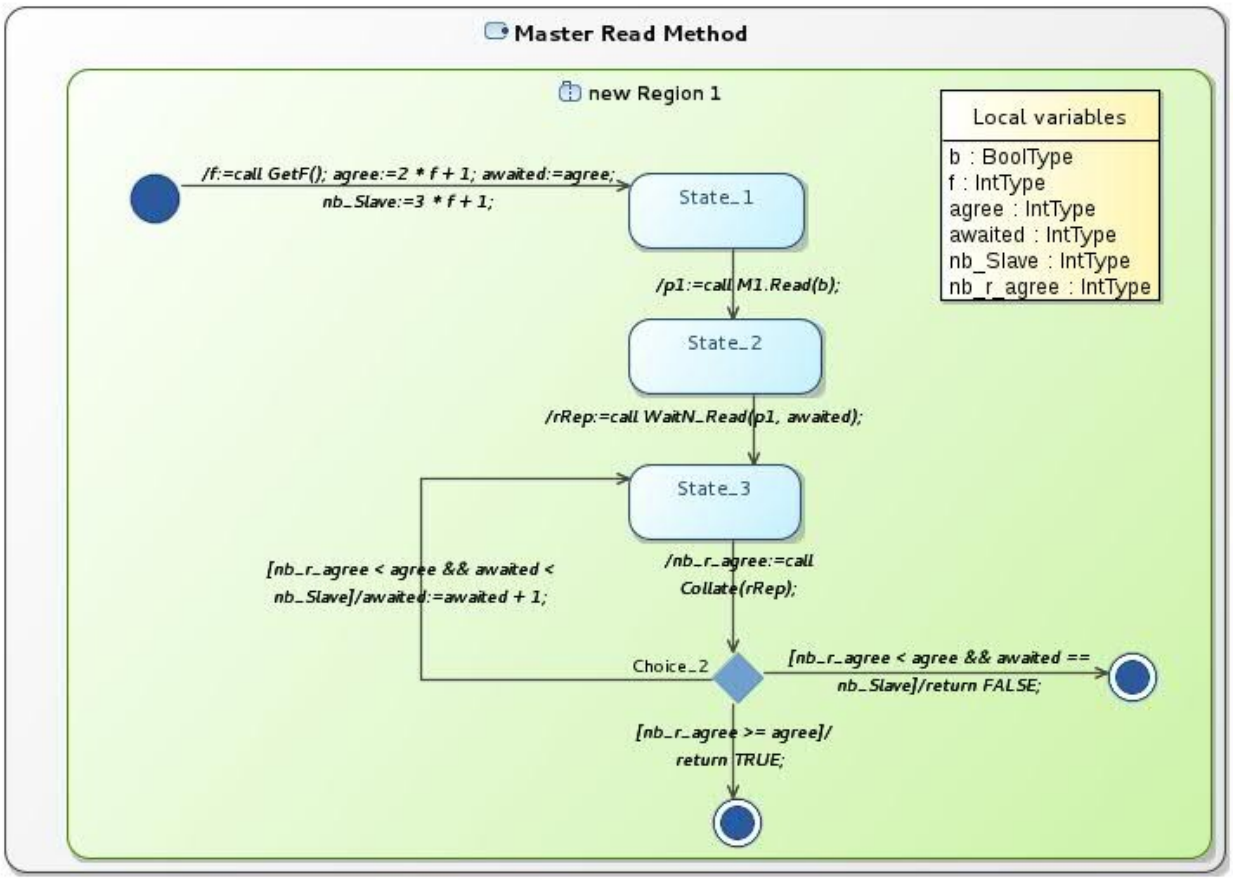
- Collate State Machine Diagram of Master Component:



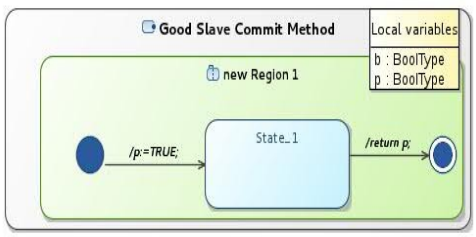
- Write State Machine Diagram of Master Component

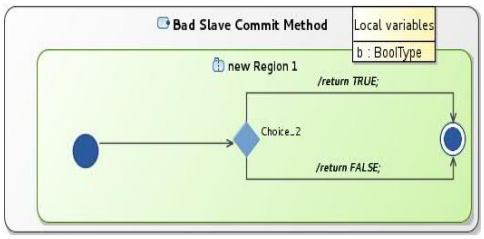


- Read State Machine Diagram of Master Component

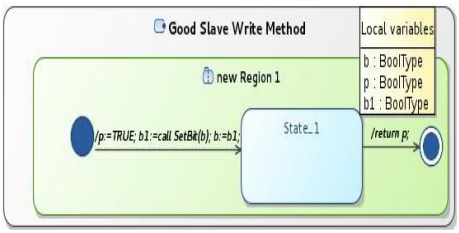


- Commit State Machine Diagram of a Slave Component

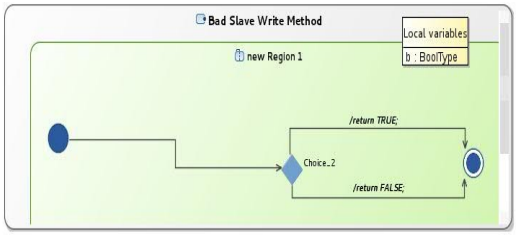




- Write State Machine Diagram of a Slave Component



•



- Read State Machine Diagram of Slave Component