



# A graphical specification environnement for GCM component-based applications

Oleksandra Kulankhina

## ► To cite this version:

Oleksandra Kulankhina. A graphical specification environnement for GCM component-based applications. Information Theory [cs.IT]. 2013. hal-00932190

**HAL Id: hal-00932190**

**<https://inria.hal.science/hal-00932190>**

Submitted on 16 Jan 2014

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



INRIA University of Nice Sophia-Antipolis CNRS I3S

UBINET International Master

Master Thesis in Computer Science

# A graphical specification environnement for GCM component-based applications

*Author:*

Oleksandra Kulankhina

*Supervisor:*

Eric Madelaine

*Academic Advisors:*

Guillaume Urvoy-Keller

## Abstract

According to the paradigm of component-based software engineering a software system can be represented as a set of independent reusable modules which communicate with each other. The OASIS team is working on a Grid Component Model (GCM) which defines how a distributed component-based application should be designed, deployed and developed. This work is focused on the modeling aspect of GCM-based applications. First, we define a formal model for the GCM-based applications architecture. Second, we provide a formalized set of consistency constraints for the GCM-based architecture validation. The created set consists of the validation rules gathered from different sources. Finally, we implement a graphical editor for the GCM-based applications architecture and behavior specifications. It has an architecture validation module which allows to verify the formalized set of constraints.

August 2013

Academic Year 2012/2013

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Introduction . . . . .	2
1.2	Motivation . . . . .	3
1.3	Objectives . . . . .	5
1.4	Contributions . . . . .	5
1.5	The structure of the report . . . . .	5
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Grid Component Model . . . . .	6
2.2	The VerCors platform . . . . .	7
2.3	UML and its application to the GCM . . . . .	8
2.4	Diagrams, semantic models and meta-models . . . . .	8
<b>3</b>	<b>Formalization</b>	<b>10</b>
3.1	The existing formal model . . . . .	10
3.2	The extended formal model . . . . .	11
3.2.1	Interfaces . . . . .	11
3.2.2	Components . . . . .	11
3.2.3	Example . . . . .	13
3.3	Validity constraints . . . . .	14
3.3.1	Namespaces . . . . .	14
3.3.2	Bindings . . . . .	15
3.3.3	Additional constraints . . . . .	17
3.4	Well Formed components . . . . .	17
<b>4</b>	<b>VCE v.3</b>	<b>19</b>
4.1	VCE v.3 overview . . . . .	19
4.2	Obeo Designer overview . . . . .	21
4.3	VCE v.3 implementation . . . . .	22
4.3.1	EMF editor for the GCM semantic models implementation . . . . .	23
4.3.2	Graphical editor for VCE Components diagrams implementation . . . . .	24
4.3.3	Implementation of specific features in VCE v.3 . . . . .	26
4.3.4	Diagrams validation . . . . .	30
<b>5</b>	<b>Related work</b>	<b>33</b>
<b>6</b>	<b>Conclusions and future work</b>	<b>35</b>
	<b>Appendices</b>	<b>38</b>
<b>A</b>	<b>VCE v.3 User guide</b>	<b>38</b>

# 1 Introduction

## 1.1 Introduction

Component based software engineering (CBSE) is a paradigm of software engineering based on two main principles: software composition and re-usability. According to the paradigm a software application might be represented as a composition of components which communicate with each other. A component is an independent module that provides some functionality. It exposes interfaces (or ports) through which the communication with its environment is performed. The same component might be used in several applications. A component may have a hierarchical structure meaning that it can be represented as a composition of other components.

The way how a component-based software is designed, developed and deployed, is defined by Component Models.

The OASIS team is working on a Grid Component Model (GCM) [22]. It is used for large-scale distributed component-based applications development. GCM is based on the Fractal Component Model [14]. GCM has the following number of features:

- First, it is a **hierarchical model** meaning that a component can be represented as a composition of the other components in GCM. The components including other subcomponents are called composite components. The smallest element of a hierarchy is called a primitive component. It provides some functionality but it has unknown internal structure.
- Second, sometimes the structure of a large scale distributed system needs to be changed at runtime. For example, when new machines with new components are added to the system and they need to establish the communication with the existing ones. GCM takes the software **reconfiguration** issue into account and provides a model for its specification.
- Third, GCM has a mechanism for the distinguishing of the components responsible for the business logic (functional concern) and the ones responsible for the control and reconfiguration issues (non-functional concern) of an application. In particular, GCM proposes a way of building non-functional components in the membrane of primitive or composite components, in charge of controlling the behaviour or the architecture of the functional components. Such components can provide an autonomous management of the GCM application.
- Finally, the communication model in GCM is adapted to the distributed applications needs. The components can communicate **asynchronously** in GCM which is not the case for Fractal. The communication is performed in a form of methods invocation. The components send the methods invocations with the sets of parameters values to each other. GCM supports the following types of communication models: one-to-one, one-to-many and many-to-many. It allows to define the multicast and gathercast communications of a distributed application.

The GCM-based applications management includes a lot of aspects: from the specification and validation of the architecture and behavior to the implementation, deployment and executing of the GCM-based applications. The architecture of a GCM-based application can be specified as a set of Component diagrams. They illustrate the components of a system and communications between them. A Component diagram reflects the static structure of a GCM application. The behavior of the primitive components can be described as UML State Machine

[10] diagrams. The behavior model of the whole system is generated in a form of parametrised networks of synchronized automata pNets[13] from the architecture and primitive components behavior specification; this behaviour model can then be used for proving behaviour properties of the application, using model-checking tools. The real components are generated from the GCM-based application architecture and behavior specification. Finally, they can be deployed and executed.

The main challenge in the management of distributed large-scale applications is to ensure that an application will be executed safely on thousands of heterogeneous machines with respect to the hierarchical structure of the components, asynchronous communications between them and the reconfiguration issue. For this purpose, first, the correctness of the specification of a distributed system static structure should be validated. For example, we need to check if there is no communication defined between two components which are unreachable for each other. Second, the generated behavior specification should be validated in order to check the dynamic properties of a distributed system. For example, in order to ensure that an application will behave correctly and that correct behaviour will be preserved in the case of reconfiguration.

This work is focused only on the specification of the GCM-based applications static properties. In particular, it is focused on the following two aspects:

1. Define a formal model of the GCM-based applications architecture and its static semantics validation.
2. Creation of a graphical editor allowing the end-users to specify the architecture of GCM-based applications and the behavior of primitive components.

First, a formal model of GCM architecture and a set of static semantics validation rules have already been developed in OASIS team[12]. However, they do not take into consideration the non-functional aspects of the GCM-based architecture. In this work we extend the existing formal model of the GCM-based applications architecture and the set of static semantics validation constraints with respect to the non-functional aspects of the GCM.

Second, we want to provide an end-user with a tool for specification of both GCM-based applications architecture and behavior. The tool should be easy-to use. It should allow a user not only to design the architecture and behavior but also to validate the static semantics constraints. A tool which partially solves this issue already exists. It is called VerCors[23] platform. It is distributed in a form of Eclipse[4] plugins. VerCors platform has a module called VCE v.2 which represents a graphical editor for the GCM Components diagrams. It has an engine for the static semantics constraints validation. However, the VerCors platform has a number of drawbacks. First, it does not provide any instruments for the components behavior specification. Second, the set of static semantics constraints validated in VCE v.2 does not include a number of essential rules. Finally, VCE v.2 is based on an old version of the Topcased[9] platform which is not maintained anymore and VCE v.2 does not work on the latest versions of Eclipse. In this work, we replace the existing VCE v.2 with a new graphical designer called VCE v.3, based on Obeo Designer [6] technology. It has an editor for the Components diagrams integrated with the UML State Machine diagrams editor for the primitive components behavior specification. The static semantics constraints validation can be performed in VCE v.3.

## 1.2 Motivation

The development of a distributed component-based system is a complicated process because a lot of aspects and issues should be taken into consideration. For example, the evolving het-

erogeneous environment of an application places demand on the reconfiguration capabilities of a software system. Also, the asynchronous communication between components leads to the possibility of deadlocks and incorrect system behavior. That is why, the general idea of the project dedicated to GCM is to provide an end-user with a simple workflow for the creation of safe GCM-based applications. The workflow should include the following steps:

1. A user specifies the architecture and behavior of a distributed software using a set of graphical editors for the GCM Components diagram, UML Class and State Machine diagrams.
2. The specified architecture and behavior correctness is checked: architecture correctness is typing and well-formedness of the application structure. Behavior correctness is about the dynamic behavior, in terms of deadlock freedom and temporal logic properties, of the application.
3. The architecture is transformed from its graphical representation into a text file. The text file contains the description of the architecture in the Architecture Description Language (ADL)[22]. ADL is an XML-based format. The ADL file must already contain the correct architecture because it was validated at the previous step. Ultimately, other part of the application implementation (source code) should also be generated, in particular the code implementing its behavior, namely the interaction between its components (but this is not part of this work).
4. The ADL file is given as an input to the Components Factory which automatically generates the components implementation from their ADL description.
5. The generated components are executed.

This work is focused on the first two steps and partially on the third one. A number of tools and theoretical models for the first three steps implementation have already been developed. In particular:

- A formal model of GCM architecture with a set of static validation constraints[12].
- A formal model of GCM behavior[12].
- A VerCors platform which has graphical a editor for the GCM-based architecture specification. It has an engine for the architecture static validation[23].
- A tool that generates an ADL description of an architecture defined in VerCors.

However, the existing theoretical models and tools have a number of issues which do not make the given workflow feasible. Namely:

- The existing formal model of GCM architecture does not contain the description of non-functional aspects of an application. As a result, we cannot ensure that an application will behave correctly in the case of reconfiguration.
- The existing formal model of GCM architecture is not compatible with the most recent versino of the ADL.

- VerCors has a graphical editor for the GCM-based applications architecture but it does not provide any tool for the behavior specification. As a result, the coherency of an application architecture and behavior specifications cannot be checked.
- The technology on which VCE v.2 is based is not maintained anymore.
- Some static semantics constraints are formalized in the existing set but not verified in VCE v.2.

### 1.3 Objectives

The objective of this work is to solve the issues given above. Namely:

- To define a formal model for the GCM architecture. The formal model should represent the extension of the existing one. It should contain the definition of the non-functional aspect. It should be consistent with the current version of the architecture description language. It should be precise enough in order to be used for the expression of the whole set of the GCM architecture validation constraints.
- To define a set of static semantics constraints for the GCM architecture validation. It should be consistent with the GCM specification, the current version of ADL, the existing set of constraints and the rules validated in the existing VerCors platform.
- To create a new version of VCE tool. It should have a Components diagrams graphical editor linked with the UML class and State Machine diagrams. It should provide a mechanism for the static semantics verification.

### 1.4 Contributions

As the result of the internship, the following contributions were done:

- The existing formal model of GCM architecture was extended with respect to the non-functional aspect of GCM architecture and the current version of architecture description language.
- The GCM architecture validation rules were gathered from different sources into a consistent formalized set of constraints.
- A new tool for the GCM architecture and behavior specification and verification was developed. It has a graphical editor for the GCM architecture linked with editors for UML State Machine and UML Class diagrams.
- The validation of the static semantics constraints was implemented in the created tool.
- The tutorial explaining how to use the new graphical editors was created.

### 1.5 The structure of the report

In Section 2, we give an informal description of the background, namely the GCM, the VerCors platform, and modelling vocabulary. Section 3 contains our extension of the formalization of GCM, and the full set of correctness constraints. Section 4 presents our work on the VCE editors, their implementation with Obeo Designer, and the current state of the correctness constraints implementation. We conclude the report with sections on related work, conclusion, and future work.

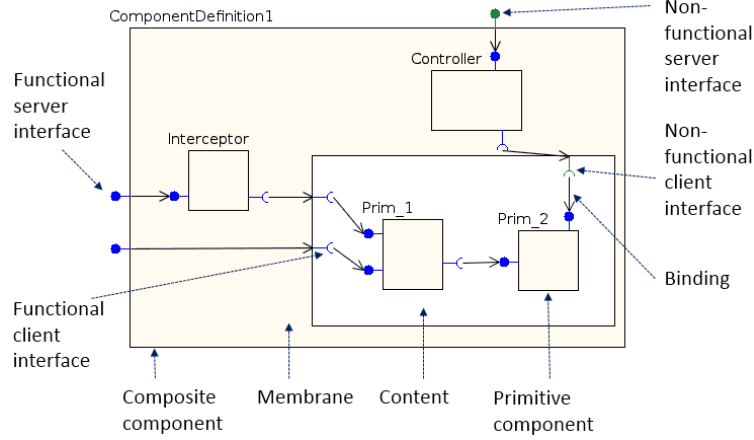


Figure 1: An example of a GCM-based architecture

## 2 Background

### 2.1 Grid Component Model

According to the Grid Component Model, a software architecture can be represented as a composition of components, interfaces and bindings. Figure 1 illustrates an example of a GCM-based application architecture.

There are two types of components in the GCM: primitive and composite ones. A primitive component is depicted as a gray rectangle. It encapsulates some business code. It has a list of methods, which it can execute. In some way it represents a black box component meaning that its content cannot be decomposed. On the contrary, a composite component's content can be decomposed and, as a result, we know its internal structure. A composite component in GCM has two main features:

1. First, it contains subcomponents (primitive or composite ones). In other words, it represents a composition of other components. That is why GCM is considered to be a hierarchical model
2. Second, it is separated into two parts: membrane (gray part) and content (white part). A content contains the nested components which perform the business logic functionality. A membrane includes subcomponents which are responsible for everything besides business logic: control, monitoring, components' structure reconfiguration, etc. This mechanism ensures the separation of functional (business logic) and non-functional (control) concerns of GCM.

The communication between components is performed in the form of methods invocation. In order to interact with their environment components use interfaces. An interface has a number of properties. Some of them are given below:

1. Role. An interface can have either client or server role. Client interfaces emit the methods invocations. Server interfaces receive them.
2. Nature. An interface can have either functional or non-functional nature. The functional interfaces accept or emit the invocations of the business-logic methods. The non-



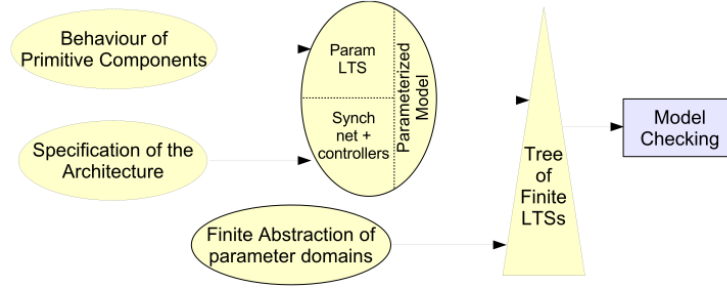


Figure 2: The principle of VerCors platform

functional ones serve for the non-functional aspects such as the components' life-cycle control, content management, etc.

3. Cardinality. If a server interface can accept several method calls at the same time from several client components, then it is called a gathercast interface. A client interface which can emit several method calls at the same time (to several target components) is called a multicast interface.
4. A list of methods. A client interface must have a list of methods that it can invoke. A server interface must have a list of methods that can be invoked through it.

The fact that two components communicate with each other is illustrated with a black arrow called a binding that goes from one server interface to another one (client).

## 2.2 The VerCors platform

VerCors is a platform for the specification, analysis, verification and validation of the GCM-based applications architecture and behavior. The principle of VerCors platform is illustrated at Figure 2. First, a user specifies the architecture of a GCM-based application and the behavior of the primitive components. Then, from these descriptions the behavioral model of an application is generated in a form of pNets. The behavioral model can be also represented as an assembly of labeled transition systems (LTs) which can use parameters. Then, the parametrized behavioral model is transformed into a language used by a Model Checker. Finally, the Model Checker verifies the correctness of the model and in the case of detected errors it provides their description.

The VerCors platform is not completely implemented yet. It does not have any editor for the primitive components behavior specification but it has an editor for the GCM-based architecture. It is called VCE v.2. Not only the specification but also the static validation of an architecture can be performed in VCE v.2. VCE is distributed as a set of Eclipse plugins. It provides a number of tools for creation of composite and primitive components, interfaces and bindings. A composite component is divided into a content and a membrane. For a primitive component a user can define a content class. It is a class containing the business code of a primitive component. The interfaces can have server or client roles, functional or non-functional natures. VCE allows to define one-to-one, one-to-many, many-to-one, and many-to-many communications between the components. A set of methods exposed by an interface cannot be defined in VCE. Instead, a user can set a "signature" of a GCM interface. Signature is a full name of a Java interface that has the list of methods exposed by a GCM interface.

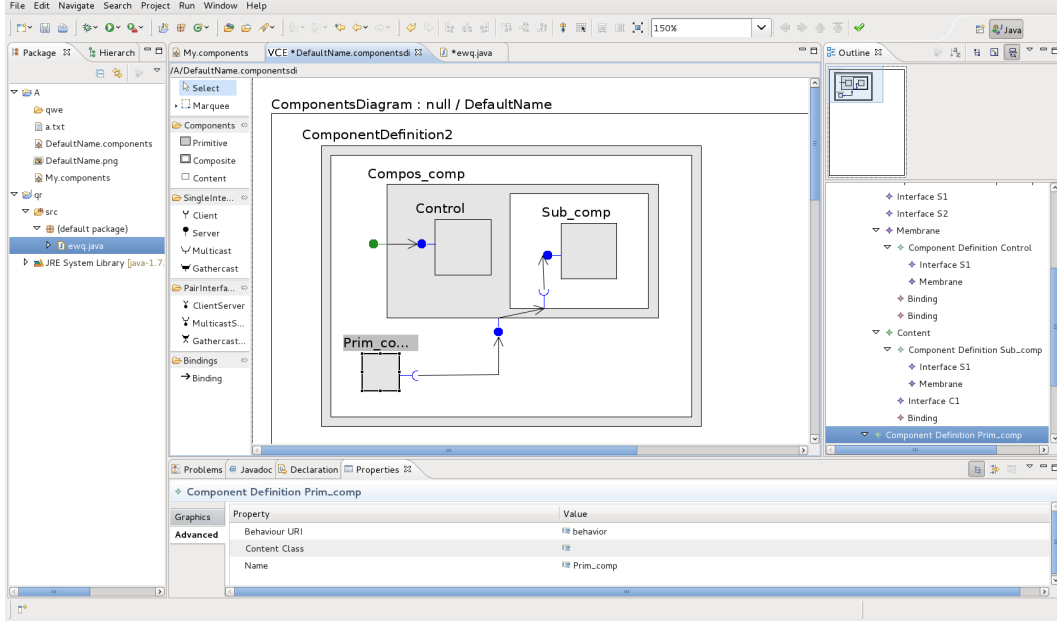


Figure 3: Screenshot of VCE-v2

Using VCE, a user can check the correctness of a GCM-based software architecture. The errors found in the architecture are depicted on a VCE Components diagram. The corresponding text description is provided. The validation rules in VCE are expressed in the Object Constraint Language (OCL [20]).

A VCE Components diagram can be exported as an image or an XML-based file.

### 2.3 UML and its application to the GCM

The Unified Modeling Language (UML) is a modeling language for the specification of various aspects of software design. UML allows to describe architecture and behavior of a system in a form of a set of diagrams.

As a result of a number of studies[11], it was proposed to use UML Class and State Machine diagrams for the specification of GCM-based applications.

UML Class diagrams illustrate classes and dependencies between them. It is a very powerful mechanism because it allows specify a lot of details. We do not need all the elements of a Class diagram. We want to use UML classes in order to specify some properties of the GCM interfaces. It was proposed to attach the UML classes to the GCM interfaces. In this case, the set of methods of a UML class corresponds to the one exposed by the corresponding GCM interface.

UML State Machine diagrams illustrate a sequence of a system's states and transitions between them. A system goes from one state to another one depending on the specified conditions. The UML State Machine diagrams can be used for the description of the GCM primitive components behavior.

### 2.4 Diagrams, semantic models and meta-models

A huge part of this work is based on three key notions: diagram, semantic model and meta-model. That is why it is important to distinguish them. Diagram is a graphical representation

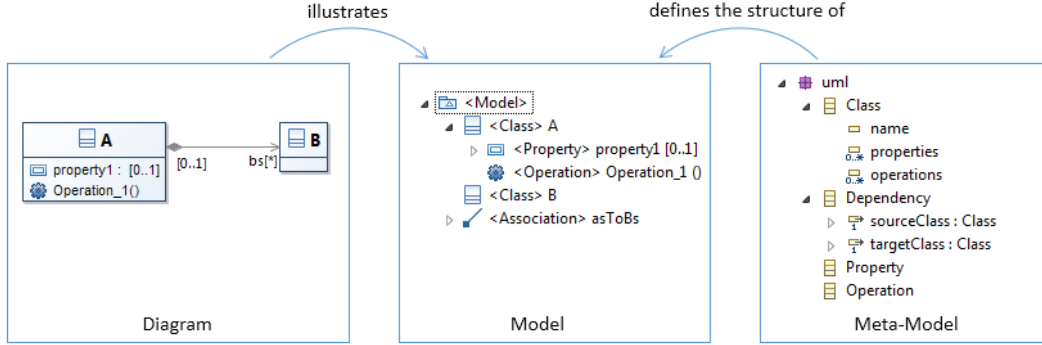


Figure 4: An example of a Class diagram, the corresponding model and meta-model

of a model. Semantic model is a set of semantic elements. Meta-model is something that defines the structure of a semantic model. It defines what elements it can contain, their properties and dependencies between them.

For example, a UML Class diagram is a picture with rectangles illustrating classes and lines illustrating dependencies between them. The semantic model is a hierarchy of semantic elements depicted by the diagram. It contains the set of classes with their properties (attributes), operations (methods) and dependencies between them. Meta-model in this case consists of a class, a property, an operation and other definitions. A meta-model says that a class can have a name, a set of properties and operations. A property can have a name and a type. Classes can be connected by dependencies, etc. Figure 4 illustrates this example.

### 3 Formalization

One of the goals of this work was to define an extended set of consistency constraints for the GCM-based applications architecture verification. This also included an extension of the existing formalization of GCM component architectures. Our sources for this work were:

- The formal model defined in the research report [12], pages [8-12], and the rules attached to this model, in the form of a *well-formedness* predicate. This model deals with component hierarchy with interfaces and bindings, including collective interfaces, but no membrane, and no difference between functional and non-functional interfaces.
- The meta-model, the component architecture, and the constraints, from the previous version of VerCors tool (VCE-V2), including non-functional interfaces, and the separation between content and membrane of composite components.
- The extension of the initial GCM model done by Paul Naoumenko PhD thesis [19, 18]. This was the seminal work introducing the membrane in GCM components, separating concerns between functional and non-functional aspects (for interfaces and for components), and giving rules for introducing components, interfaces and bindings in the membrane.

We extended the formal model from [12] to include non-functional concerns, and the content/membrane objects in the architecture. We had to modify some of the constraints gathered from the 3 different sources, essentially with respect to the non-functional concerns. Some of the constraints were similar, some of them contradicted each other. Eventually, after several iterations of discussions inside the team, we managed to provide a consistent set of constraints for the verification of GCM-based applications architecture, and we formalized fully the corresponding well-formedness predicates.

In this section we start with the definition of the existing and of our extended formal model, then we describe (informally and formally) the full set of constraints, explaining the choices we made when necessary.

#### 3.1 The existing formal model

We start with the formal model from [12], representing the GCM architecture. It includes the definitions of the server and client interfaces, primitive and composite components and bindings. The components' definition does not include the membrane. The existing formal model is provided below:

$$SItf ::= (Name, Card, MSignature_i^{i \in I})_S \quad (1)$$

$$CItf ::= (Name, Card, MSignature_i^{i \in I})_C \quad (2)$$

$$Itf ::= SItf \mid CItf \quad (3)$$

$$Prim ::= Cname < SItf_i^{i \in I}, CItf_j^{j \in J}, M_k^{k \in K} > \quad (4)$$

$$Compos ::= Cname < SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, Binding_l^{l \in L} > \quad (5)$$

$$Comp ::= Prim \mid Compos \quad (6)$$

$$Binding ::= (QName, QName) \quad (7)$$

$$QName ::= This.Name \mid CName.Name \quad (8)$$

A binding is described as a couple of names. Each name consists of two parts. The first part is the name of the sub-component or *This*. The second part is the name of the interface.

## 3.2 The extended formal model

Now we add the non-functional aspects to the model, and the membrane structure.

### 3.2.1 Interfaces

First, a notion of an interface *nature* must be formalized. An interface nature indicates if it is a functional (F) or a nonfunctional (NF) interface.

$$Nature ::= F \mid NF \quad (9)$$

Now a server and a client interfaces structures are extended as <sup>1</sup>:

$$SItf ::= (Name, Card, MSignature_i^{i \in I}, Nature)_S \quad (10)$$

$$CItf ::= (Name, Card, MSignature_i^{i \in I}, Nature)_C \quad (11)$$

### 3.2.2 Components

In our extended model, primitive components have a membrane, and composite components have a membrane and a content.

One tricky point was to decide how to represent in our formal model the relations between interfaces of the component and those inside the content and the membrane. Paul Naoumenko proposed to duplicate some of the interfaces. He also introduced some rules for the interfaces that are duplicated. As a result, it turns out that most of the interfaces appear in the ADL file twice. If we duplicate all the interfaces inside the model we will get a heavy formal model with quite complicated rules. Hence, we propose not to follow all the ADL extensions in the formal model directly but to do the following:

1. include all the external functional and non-functional interfaces in the component's definition;
2. include all the internal functional and non-functional interfaces in the content's definition;
3. do not to include any interfaces in the membrane's definition but compute them using a special function *Symmetry*. For a client interface it returns the corresponding server one. For a server interface it returns the corresponding client one. In order to compute the interfaces of a membrane we should take a set of the interfaces of a component which contains the membrane and a set of interfaces of its content;
4. in order to be able to compute the set of the interfaces a membrane should have a reference to the component which contains it.

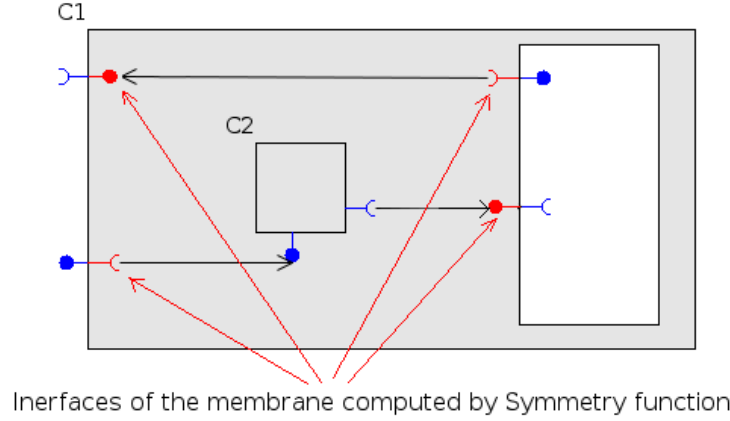


Figure 5: Symmetry function

For example, figure 5 illustrates GCM architecture with a set of additional interfaces computed by the *Symmetry* function.

Taking into consideration all the modifications mentioned above, the following formal definition of a membrane is proposed:

$$Membr ::= < Comp_k^{k \in K}, Binding_l^{l \in L}, Interc_n^{n \in N}, CompRef > \quad (12)$$

Here *CompRef* represents a reference to the component that contains the membrane.

The interceptors are components that have special responsibilities; they are components within the membrane that are placed behind a functional service interface, and that work on the functional data-flow going through this interface. The other components within the membrane are non-functional components, we will see later their role and the corresponding constraints. But both interceptors and non-functional components can be either primitive or a composite components. In a membrane's definition we just put the components that represent the interceptors in a separated set.

The example in Figure 6 shows an interceptor named “Inter\_1”, bound behind the server interface “S1” of the composite component, and to the internal interface “C1” of the content. “S1,S2,C2,C1” constitute a functional chain of interfaces and bindings that all functional requests arriving on “S1” will follow. The interceptor also has a non-functional interface “Inter\_1.C1” that is bound to an interface of the non-functional component ControlPrim, within the membrane.

The membrane's definition does not explicitly include interfaces. They are computed by *Symmetry* function with the following definition:

$$\begin{aligned} Symmetry((Name, MSignature_i^{i \in I}, Nature)_S) &::= \\ &::= (Name, MSignature_i^{i \in I}, Nature)_C \\ Symmetry((Name, MSignature_i^{i \in I}, Nature)_C) &::= \\ &::= (Name, MSignature_i^{i \in I}, Nature)_S \end{aligned} \quad (13)$$

Now we can introduce:

---

<sup>1</sup>In this report, we do not include the formalization of *Collective Interfaces*, that were already formalized in [12]. Adding them (with the corresponding constraints) in the extended model is straightforward. We have simply left the *Card* element in the interface object.

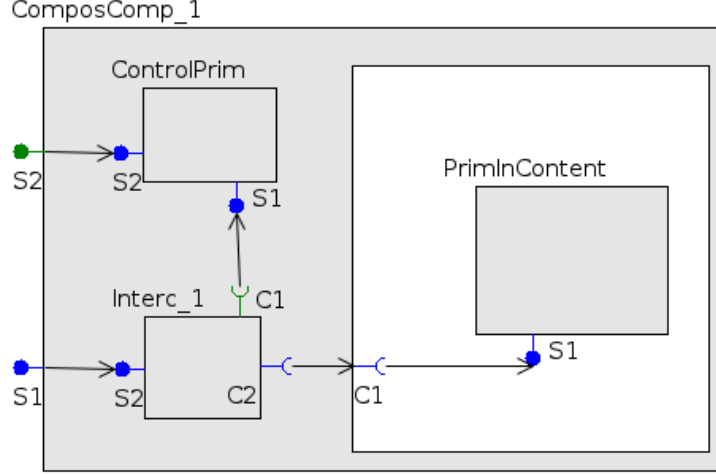


Figure 6: Example of a GCM Diagram, drawn with VCEv2

The modified representation of a primitive component which includes a componentized membrane <sup>2</sup>:

$$Prim ::= Cname, < SItf_i^{i \in I}, CItf_j^{j \in J}, M_k^{k \in K} Membr > \quad (14)$$

Now we need to introduce a content in the description of a composite component. A content may contain, first, interfaces which are indeed the internal interfaces of a composite component, second, other components and, finally, bindings.

Finally, a composite component has a content and a membrane. They are compulsory elements now. All the components and bindings that were included in the composite component's definition in the previous model, should be now included either in the membrane or in the content.

$$Cont ::= < SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, Binding_l^{l \in L} > \quad (15)$$

$$Compos ::= Cname, < SItf_i^{i \in I}, CItf_j^{j \in J}, Membr, Cont > \quad (16)$$

The component definition remains the same as in the previous model.

$$Comp ::= Prim \mid Compos \quad (17)$$

The definition of Bindings is the same as (8) except the fact that now *This* in a *QName* may corresponds not only to a component but to a membrane or a content.

### 3.2.3 Example

We provide here an example of a toy GCM composite component, with its full formal model.

Figure (6) illustrates an example of a GCM diagram. The corresponding formal model is given below. By convention, in this formal model the names of variables representing the

<sup>2</sup>Note that components in a primitive membrane cannot be bound to the functional content of the primitive. Despite this particularity they are useful, and can be either Interceptors, standard Fractal controllers, or user-defined non-fonctionnal components.

interfaces are given in the form of  $X - Y - Z$ . Here  $X = s$  if it is a server interface and  $X = c$  if it is a client interface.  $Y$  is some name derived from the component's name (e.g. "cc" for the *ComposComp\_1*).  $Z$  is the number of the interface in the set of the component's interfaces.

$$\begin{aligned}
composite &= \text{ComposComp\_1} < \{s - cc - 1, s - cc - 2\}, \{\}, membr, cont > \\
s - cc - 1 &= (S1, \{\}, F)_S \\
s - cc - 2 &= (S2, \{\}, NF)_S \\
membr &= < Comp_k^{k \in \{mc1\}}, Binding_l^{l \in \{mb1, mb2, mb3, mb4\}}, Interc_n^{n \in \{mi1\}}, composite > \\
Comp_{mc1} &= \text{ControlPrim} < \{s - cp - 1, s - cp - 2\}, \{\}, < > > \\
s - cp - 1 &= (S1, \{\}, F)_S \\
s - cp - 2 &= (S2, \{\}, F)_S \\
Interc_{mi1} &= \text{Interc\_1} < \{s - i - 1\}, \{c - i - 1, c - i - 2\}, \{\}, < > > \\
s - i - 1 &= (S2, \{\}, F)_S \\
c - i - 1 &= (C1, \{\}, NF)_C \\
c - i - 2 &= (C2, \{\}, F)_C \\
Binding_{mb1} &= (This.S1, Interc\_1.S2) \\
Binding_{mb2} &= (This.S2, ControlPrim.S2) \\
Binding_{mb3} &= (Interc\_1.C1, ControlPrim.S1) \\
Binding_{mb4} &= (Interc\_1.C2, This.C1) \\
cont &= < \{\}, \{c - cc - 1\}, Comp_k^{k \in \{cc1\}}, Binding_l^{l \in \{cb1\}} > \\
c - cc - 1 &= (C1, \{\}, F)_C \\
Comp_{cc1} &= \text{PrimContent} < \{\}, \{c - pc - 1\}, \{\}, < > > \\
c - pc - 1 &= (S1, \{\}, F)_S \\
Binding_{mb4} &= (This.C1, PrimContent.S1)
\end{aligned}$$

The membrane definition does not include any interfaces. Its interfaces are computed using the *Symmetry* function. The function is applied to the component's interfaces and content's interfaces. As a result, we can create bindings that connect the membrane's interfaces even if they are not explicitly defined. For example,  $Binding_{mb1}$  goes from  $This.S1$  to  $Interc\_1.S2$ . Here  $This$  corresponds to the membrane but the membrane does not contain any explicitly defined interfaces. The interface  $S1$  of the membrane is a client interface. It is the result of the *Symmetry* function. The function was applied to the server interface  $S1$  of the composite component *ComposComp\_1*.

### 3.3 Validity constraints

We describe now the set of constraints that define a *well-formed* GCM model. We present them in 3 different subsets depending on their nature, and for each of them discuss their informal meaning and give a formal definition.

#### 3.3.1 Namespaces

The first set of constraints deals with naming conflicts:



- Constraint: all the components at the same level of hierarchy must have different names. At the same time, a membrane and a content of the same composite component are considered as different namespaces.

This rule is defined using an auxiliary predicate *UniqueItfNames*, that will be used both in the WF predicate of membranes and of contents.

$$UniqueItfNames(Itf_i^{i \in I}) \Leftrightarrow \forall i, i' \in I. i \neq i' \Rightarrow Name(SItf_i) \neq Name(SItf_{i'}) \quad (18)$$

- Constraint: all the interfaces of a component must have different names. Here again, membranes and contents have separated namespaces, so this constraint will be checked separately, for the external interfaces of a component, and for the internal interfaces of a membrane and of a content:

It uses the auxiliary predicate *UniqueCompNames*:

$$UniqueCompNames(Comp_i^{i \in I}) \Leftrightarrow \forall i, i' \in I, i \neq i' \Rightarrow CName(Comp_i) \neq CName(Comp_{i'}) \quad (19)$$

### 3.3.2 Bindings

The second set of constraints deals with compatibility of interfaces related by a binding. This includes role, typing, and nature compatibility: Most of these rules will use an auxiliary function *Get*, able to retrieve an Interface object from its name, given its name and its “container” (either a component, a membrane, or a content):

$$Get : QName \times Comp \cup Membr \cup Cont \cup Interc \rightarrow Itf \quad (20)$$

If *Get* function is applied to a membrane then it searches for an interface returned by the *Symmetry* function applied to the corresponding component that contains the membrane and its content.

Conversely, some rules use the *Parent* auxiliary function, that recover the container (component, membrane or content) of an interface:

$$Parent : (Itf_i^{i \in I} \cup Comp_j^{j \in J}) \times context \rightarrow Membr \cup Cont \cup Comp_k^{k \in K} \quad (21)$$

- Role constraint: in the original formal model, we had a rule for bindings between interfaces of sibling components that must go from a client interface to server interface; and of delegation bindings, going from an service interface of a composite to a service interface of one of its sucomponent, or from a client interface of a subcomponent to a client interface of the composite.

This constraint has changed in the extended model: bindings still go from server to client interfaces between sibling components (within membrane or within content). But this is also true for delegation bindings, thanks to the *Symmetry* function, making the role constraint uniform:

$$BindingRoles((Src, Dst)) \Leftrightarrow Get(Src).role = C \wedge Get(Dst).role = S \quad (22)$$

- Typing constraint: a binding must bind interfaces of compatible types. The compatibility of interfaces means that for each method of a client interface there must exist an adequate

corresponding method in the server interface. In other words, when a client interfaces is connected to a server interface and it want to call some method, then this method must actually exist on the server interface.

In general, an adequate corresponding method need not have exactly the same signature than the one required, but can use any subtyping or inheritance pre-order available in the modeling language. We denote  $\leq$  such an order between interface signatures in the following constraint:

$$BindingTypes((Src, Dst)) \Leftrightarrow Get(Src).MSignature \leq Get(Dst).MSignature \quad (23)$$

- Nature constraint: this is a rule imposing a separation of concerns between functional and non-functional aspects. We want functional interfaces to be bound together (only functional requests will be going through these), and non-functional nrequests to go through non-functional interfaces. This is simple to impose in the content of composite components, but a little more tricky in the membrane because of the specific status of interceptors.

The solution is to qualify as functional all components in the content, but also to interceptors, while all other components in the membrane are non-functional. Then interfaces are declared functional or non-functional, and from this we compute for each interface a control level ranging from 0 to 2, where zero means functional. Then compatible interfaces are either both “0”, or both greater than “0”.

This constraint is significantly simpler and more intuitive, and certainly more coherent, than those that were informally expressed in [19].

$$BindingNature((Src, Dst)) \Leftrightarrow \quad (24)$$

$$CL(Src) = CL(Dst) = 1 \vee (CL(Src) > 1 \wedge CL(Dst) > 1) \quad (25)$$

Where the ControlLevel function is defined as:

$$CL(X \in Comp_j^{j \in J}) ::= \begin{cases} 2, & \text{if } parent(X, context) = Membr \wedge X \in Comp(Membr) \\ 1, & \text{if } parent(X, context) = Membr \wedge X \in Interc(Membr) \\ 1, & \text{else} \end{cases}$$

$$CL(X \in Itf_i^{i \in I}) ::= \begin{cases} CL(parent(X, context)), & \text{if } Nature(X) = F \\ CL(parent(X, context)) + 1, & \text{if } Nature(X) = NF \end{cases} \quad (26)$$

- No Self Binding constraint: the source and the target interfaces of a binding cannot belong to the same component, the same content or the same membrane,

$$NoSelfBinding((Src, Dst)) \Leftrightarrow Parent(Src) \neq Parent(Dst) \wedge \quad (27)$$

- No Binding Race constraint: no two bindings that go from the same source interface can have target interfaces that belong to the same content, component or membrane.

$$NoBindingRace((Src1, Dst1)(Src2, Dst2)) \Leftrightarrow \quad (28)$$

$$Src1 = Src2 \Rightarrow Parent(Dst1) \neq Parent(Dst2) \quad (29)$$

### 3.3.3 Additional constraints

The last set of constraints gathers some rules that are not formalized in the formal model (but will be checked in the implementation).

- Primitive services: for all the methods declared in the server interfaces of a primitive component there must exist the corresponding methods definitions in the primitive component. In other words, it means that if a primitive component declares that it can execute a concrete method (by exposing it on a server interface), then it must actually have this method.

We do not formalize this method because the services methods in the implementation of a primitive component are not described in the formal model.

- Well-formed bindings: in the diagram editors of the VCE, it is checked that no binding crosses a component border. This is not a constraint that can be checked at the level of the formal model, because it must be validated before building the formal model.
- Syntactic correctness of names: in the GCM ADL names of components and interfaces must obey some lexicographic constraints, e.g. cannot contain white space characters.

We leave the precise definition of this constraint to the VCE implementation.

### 3.4 Well Formed components

As a last step, we use the previous constraints to define a well-formedness predicate, denote  $WF$ , recursively on component architecture, namely on membranes, contents, on primitive and composite components.

A **well-formed membrane** is a membrane that satisfies all the following constraints:

1. all its subcomponents and interceptors have distinct names;
2. all its subcomponents and interceptors are well-formed;
3. its bindings respect the 5 binding constraints:

BindingRoles: each binding must go from a client to a server interfaces,

BindingTypes: the source and the target interfaces of a binding have compatible types,

BindingNatures: the control level of the interfaces connected by each binding is either equal to 1 for both interfaces or more than one for both interfaces,

NoSelfBinding: the source and the target interfaces of a binding cannot belong to the same component, the same content or the same membrane,

NoBindingRaces: no two bindings that go from the same source interface can have target interfaces that belong to the same content, component or membrane.

$$WF(membr = \langle SItf_i^{i \in I}, CItf_j^{j \in J}, Comp_k^{k \in K}, Binding_l^{l \in L}, Interc_n^{n \in N} \rangle) \Leftrightarrow$$

$$\Leftrightarrow \begin{cases} UniqueCompNames(Comp_k^{k \in K} \cup Interc_n^{n \in N}) \wedge \\ \forall k \in K. WF(Comp_k) \wedge \\ \forall n \in N. WF(Interc_n) \wedge \\ \forall B \in Binding_l^{l \in L}. BindingRoles(B) \wedge BindingTypes(B) \wedge BindingNature(B) \wedge NoSelfBinding(B) \\ \forall B1, B2 \in Binding_l^{l \in L}. NoBindingRaces(B1, B2) \end{cases} \quad (30)$$

A **content** of a composite component is well-formed if it satisfies the following set of constraints:

1. all its interfaces have distinct names;
2. all its subcomponents have distinct names;
3. all its subcomponents are well-formed;
4. its bindings respect the 5 binding constraints (BindingRoles, BindingTypes, BindingNature, NoSelfBinding, NoBindingRace).

$$\begin{aligned}
WF(cont = < SItf_i^{\dot{i} \in I}, CItf_j^{\dot{j} \in J}, Comp_k^{k \in K}, Binding_l^{l \in L} >) \Leftrightarrow \\
\Leftrightarrow \left\{ \begin{array}{l}
UniqueItfNames(SItf_i^{\dot{i} \in I} \cup CItf_j^{\dot{j} \in J}) \wedge \\
UniqueCompNames(Comp_k^{k \in K}) \wedge \\
\forall k \in K. WF(Comp_k) \wedge \\
\forall B \in Binding_l^{l \in L}. BindingRoles(B) \wedge BindingTypes(B) \wedge BindingNature(B) \wedge NoSelfBinding(B) \\
\forall B1, B2 \in Binding_l^{l \in L}. NoBindingRaces(B1, B2)
\end{array} \right. \quad (31)
\end{aligned}$$

A **well-formed primitive component** is a component that satisfies the following set of constraints:

1. all its interfaces have distinct names;
2. its has well-formed membrane;

$$\begin{aligned}
WF(Cname < SItf_i^{\dot{i} \in I} \cup CItf_j^{\dot{j} \in J}, M_k^{k \in K}, Membr >) \Leftrightarrow \\
\Leftrightarrow \left\{ \begin{array}{l}
UniqueItfNames(SItf_i^{\dot{i} \in I}, CItf_j^{\dot{j} \in J}) \wedge \\
WF(Membr)
\end{array} \right. \quad (32)
\end{aligned}$$

A **well-formed composite component** is a component which satisfies the following set of constraints:

1. all its interfaces have distinct names;
2. its membrane is well-formed;
3. its content is well-formed;

$$\begin{aligned}
WF(Cname, < SItf_i^{\dot{i} \in I} \cup CItf_j^{\dot{j} \in J}, Membr, Cont >) \Leftrightarrow \\
\Leftrightarrow \left\{ \begin{array}{l}
UniqueItfNames(SItf_i^{\dot{i} \in I}, CItf_j^{\dot{j} \in J}) \wedge \\
WF(Membr) \wedge \\
WF(Cont)
\end{array} \right. \quad (33)
\end{aligned}$$

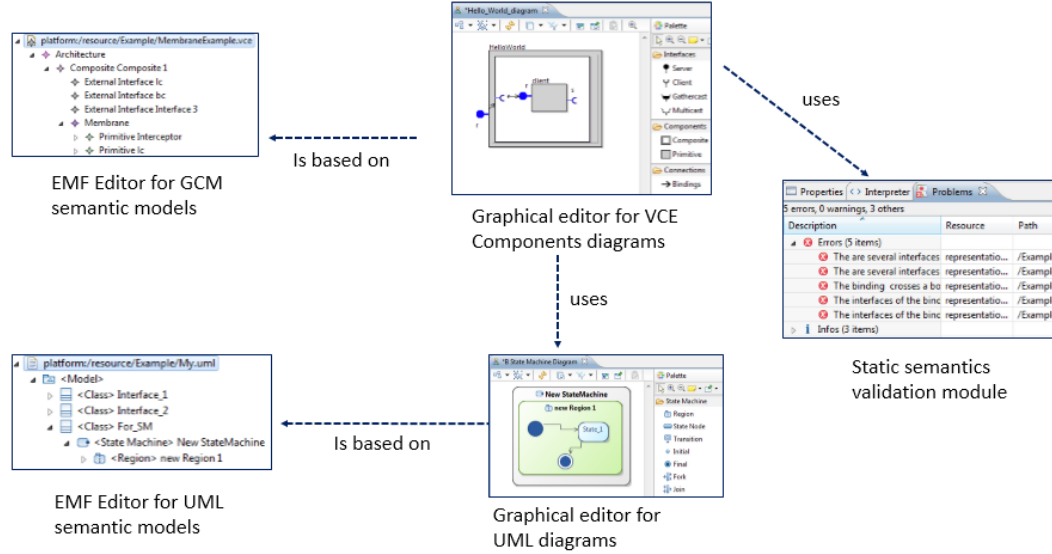


Figure 7: vce.ecore

## 4 VCE v.3

The second part of the project was focused on the creation of a new version of VCE. The updated tool should have a graphical editor for GCM Components (called VCE Component diagrams editor) linked with UML Class and State Machine diagrams graphical editors. The updated tool should have a static semantics constraints validation module.

This work was the continuation of previous work in the Oasis team, by Nicolas Le Halper [15], then by myself during my PFE project [16], exploring the possibilities of creating such editors with the open-source Papyrus platform. The conclusion was that on one side we were unable to create an inter-connected set of diagram editors that would include both a user-defined DSL, and a UML-based diagram editor; and on the other side Papyrus was poorly documented, and too instable to support such development, with frequent releases not guarantying backward compatibility.

Eventually, we managed to implement such tool. It is called VCE v.3. For this purpose we used Obeo Designer[6] technology. We created a tutorial explaining how to use VCE v.3 and a number of use-case diagrams. They can be found in the annex. This section provides the overview of the VCE v.3 features, the overview of Obeo Designer technology. Then it provides the technical details of VCE v.3 implementation.

### 4.1 VCE v.3 overview

VCE v.3 allows to create GCM architecture and UML semantic models, display them graphically on the VCE Components and UML diagrams, validate the static semantics constraints of the GCM-based architecture. Figure 7 illustrates the architecture of VCE v.3. According to it, the tool has the following five modules:

1. **EMF editor for GCM semantic model** allows to see the elements of a GCM architecture semantic model in a form of a tree. It provides tools for creation, edition and removing of GCM components, bindings and interfaces. We developed the editor.

2. **EMF editor for UML semantic model** allows to see the elements of a UML semantic model in a form of a tree. It provides tools for creation, edition and removing of UML semantic elements. It is a standard EMF editor of UML semantic models for Eclipse.
3. **Graphical editor for VCE Components diagrams** displays the Components diagrams illustrating the GCM semantic models. It provides the number of tools for the edition of GCM semantic models and VCE Components diagrams. Among the other elements of a VCE Components diagram, it displays the UML State Machines illustrating the behavior of primitive components and UML classes attached to GCM interfaces. We developed the graphical editor for the VCE Components diagrams.
4. **Graphical editor for UML diagrams** displays UML diagram and allows to edit them. It is an open source editor provided by Obeo Designer [8].
5. **Static semantics validation module** allows to validate the static semantics constraints on the VCE Components diagrams. The module was developed by us.

The GCM-based architecture specification process has the following workflow in VCE v.3:

1. A user creates a new Modeling Project. The Modeling Project will store all the files of semantic models and diagrams.
2. A user creates a new GCM architecture semantic model using an EMF editor for GCM semantic models.
3. Using Graphical editor for VCE Components diagram a user creates and edits a new VCE Components diagram based on the created GCM semantic model. A diagram cannot exist without a semantic model.
4. A user creates a new UML semantic model using an EMF editor for UML semantic models.
5. A user creates and edits UML diagrams using the EMF editor for UML diagrams.
6. Now a user can set references from a GCM model to the UML one. He can set a UML State Machine defining the behavior of a primitive component. He can attach a UML class to a GCM interface. The referred UML elements can be displayed on a VCE Components diagram.
7. A user can validate the static semantic constraints on the VCE components diagrams.
8. If the diagrams are valid, then the user can trigger the generation of the corresponding ADL file.

Figure 8 illustrates the structure of the resulting Modeling Project and the dependencies between its entities.

In addition, not only UML Class and State Machine but also Use case, Deployment and other UML diagrams can be created in VCE v.3. As a result a distributed software system can be specified from several points of view.

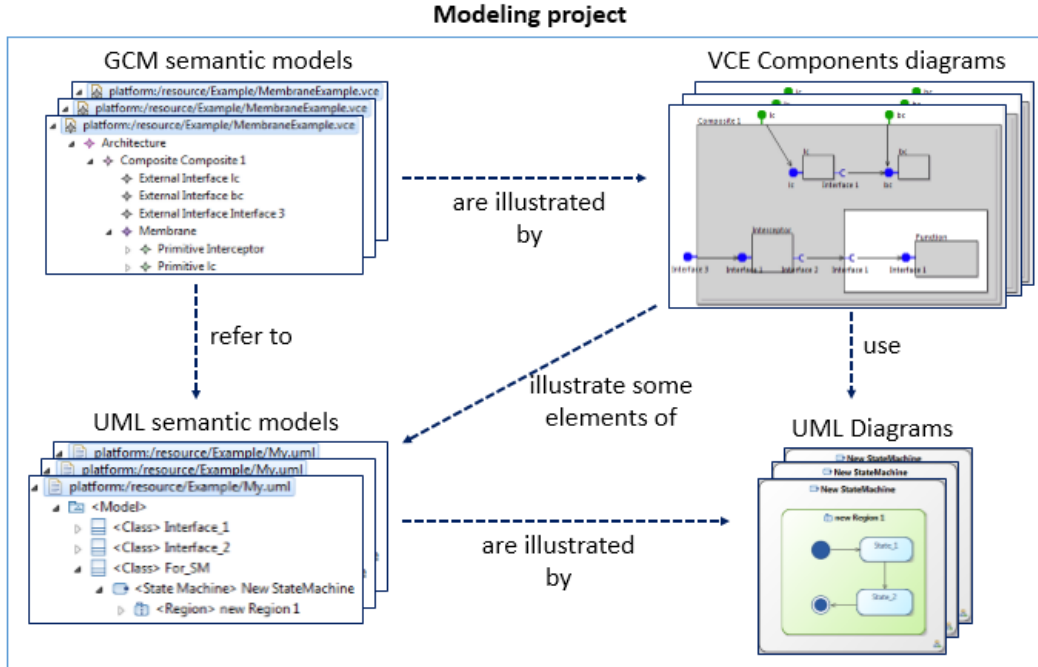


Figure 8: A Modeling Project structure in VCE v.3

## 4.2 Obeo Designer overview

Obeo Designer is an Eclipse-based software for the graphical editors creation. It is based on the Graphical Modeling Framework (GMF)[5] and Eclipse Modeling Framework (EMF)[2] technologies. It uses Aceleo[1] language for implementation of some features. This section describes the overview of a workflow of a custom graphical editor creation in Obeo Designer.

The workflow is defined in Obeo Tutorial[7] and illustrated on Figure 9. According to it, in order to develop new EMF and Graphical editors in Obeo Designer the following steps should be implemented.

- **Step 0:** a developer creates an Ecore Modeling project. An .ecore file is generated there automatically. It is an XML-based file.
- **Step 1:** in .ecore file a developer specifies the meta-model of his editor. More precisely, he defines what kind of elements will be specified in the created editor, what attributes and dependencies they will have.
- **Step 2:** from an .ecore file a developer generates the source code of the meta-model elements and two other projects: .edit and .editor. They contain the source code for a standard EMF semantic model editor.
- **Step 3:** a developer specifies the graphical editor in the following way. He creates a new .design project. It has an .odesign file. In .odesign file, a developer specifies the graphical representation of the diagrams elements, the tools available for the elements edition and the diagram validation rules. The following constructs are supported for the graphical representation of the elements: nodes, containers, bordered nodes (they are attached to the other nodes and they usually represent ports), element-based edges and relation-based

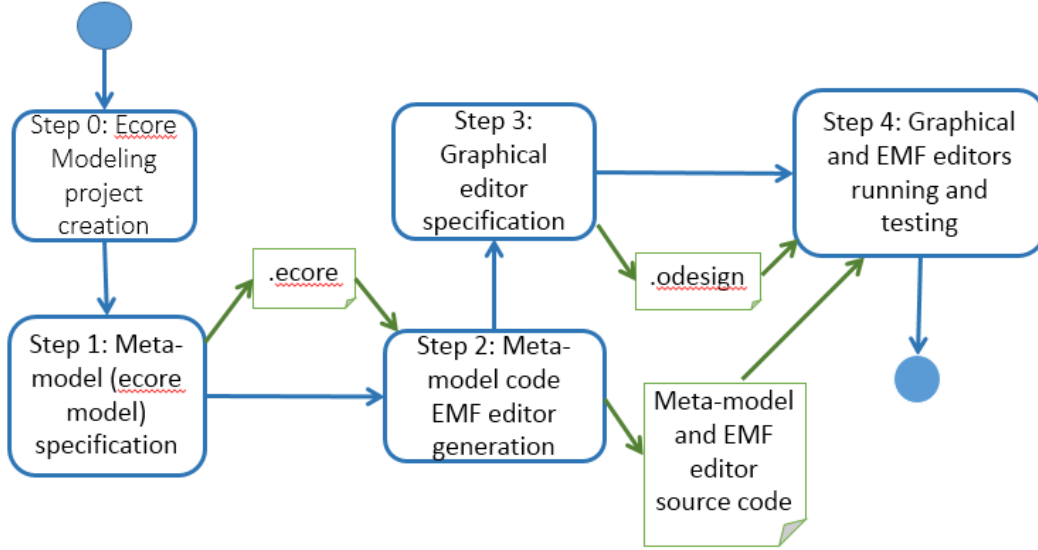


Figure 9: EMF and Graphical editors workflow creation in Obeo Designer

edges. A developer can set a style of each element. Obeo Designer provides a number of shapes and styles. It also allows to define your own custom style for some of the elements. The methods of Java classes can be called for the custom styles support. Also, elements can be imported from one .odesign file to another. A developer can specify wide range of tools: create, delete, edge reconnect tools, etc. The specified tools can call external Java classes specified by a developer. The validation rules can be expressed either in OCL or in Aceleo[REF] language. They can also call methods of Java classes.

- **Step 3:** a developer runs and tests the EMF editor and the graphical diagrams editor as a new Eclipse application.

As a result, four projects should have been created: the ecore modeling project containing .ecore model, .edit and .editor projects with a standard EMF editor source code and a .design project with a diagram editor specification in .odesign file. Each project should have a *plugin.xml* file where a developer specifies its additional properties. The four projects can be deployed and distributed as the Obeo Designer plugins. They can be installed on Obeo Designer or on Eclipse with Obeo Designer plugin installed. It is a big disadvantage of the technology because Obeo Designer is a commercial project. Hence, in order to install a graphical editor based on their technology, one needs to buy Obeo Designer.

However, it has been announced that an open-source version of Obeo Designer, called Sirius, will be released in September, 2013.

### 4.3 VCE v.3 implementation

Following the standard Obeo Designer workflow we implemented an EMF editor for the GCM architecture semantic models and an editor for VCE Components diagram. Then we integrated them with the existing editors for the UML semantic models and UML diagrams. In this section, first, we provide the implementation source structure overview. Second, we describe the implementation of an EMF editor for GCM architecture semantic models. Third, we explain the implementation of a graphical editor for VCE Components diagrams. Then, we explain how



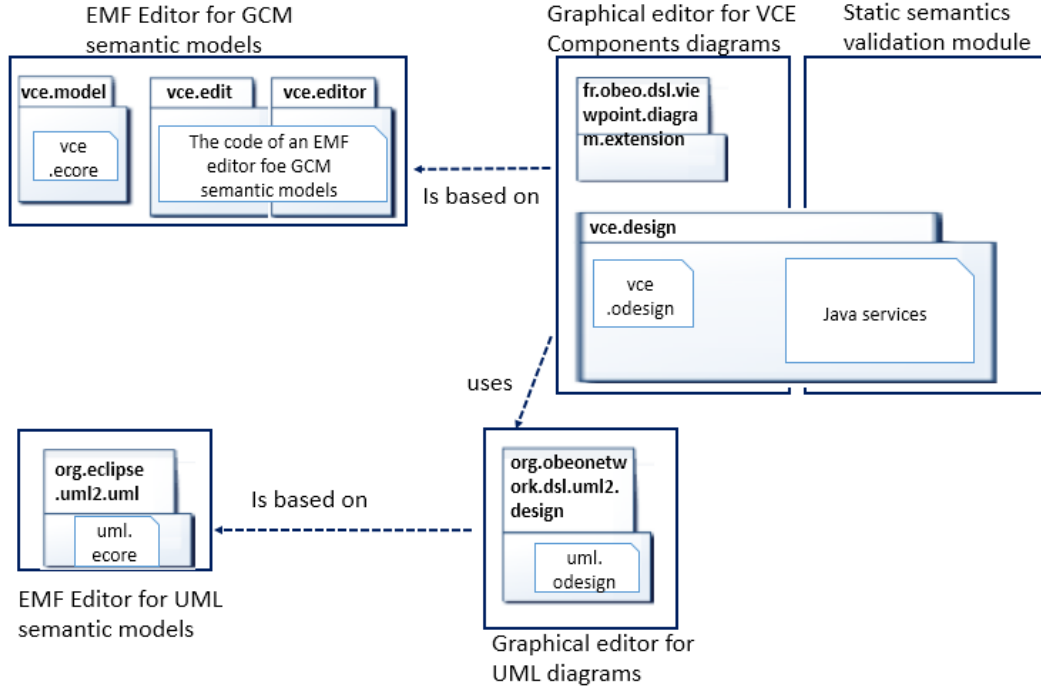


Figure 10: VCE v.3 source structure

the most tricky features were developed. Finally, we tell about the VCE Components diagrams static semantics validation implementation.

Figure 10 illustrates the source projects that lie behind the five modules of VCE v.3 illustrated at Figure 7.

VCE v.3 consists of six plugins.

**vce.model.** The plugin contains the .ecore meta-model of the VCE Components diagrams graphical editor.

**vce.edit and vce.editor.** The plugins contain a set of classes implementing the standard EMF editor for VCE Components models.

**vce.design.** The plugin implements the graphical editor of VCE Components diagrams.

**org.obeonetwork.dsl.uml2.design.** The plugin implements a graphical editor of UML diagrams. It was downloaded from the git repository of Obeo community[8].

**fr.obeo.dsl.viewpoint.diagram.extension.** The plugin helps to change a picture of an interface depending on which side of a component it is attached to.

VCE.v3 also uses all the standard plugins distributed with Obeo Designer.

#### 4.3.1 EMF editor for the GCM semantic models implementation

The EMF editor for GCM semantic models is based on the vce.ecore file containing the meta-model of the GCM semantic model implementation. Figure 11 illustrates the vce.ecore meta-model. Here, Architecture is the root element of the diagram. All the elements of a diagram or their parents belong to the Architecture. Architecture is a mandatory element of a diagram. Each diagram can have only one Architecture. An architecture can contain components and bindings.

Component is an abstract element. Its instances cannot be created. It has two parameters:

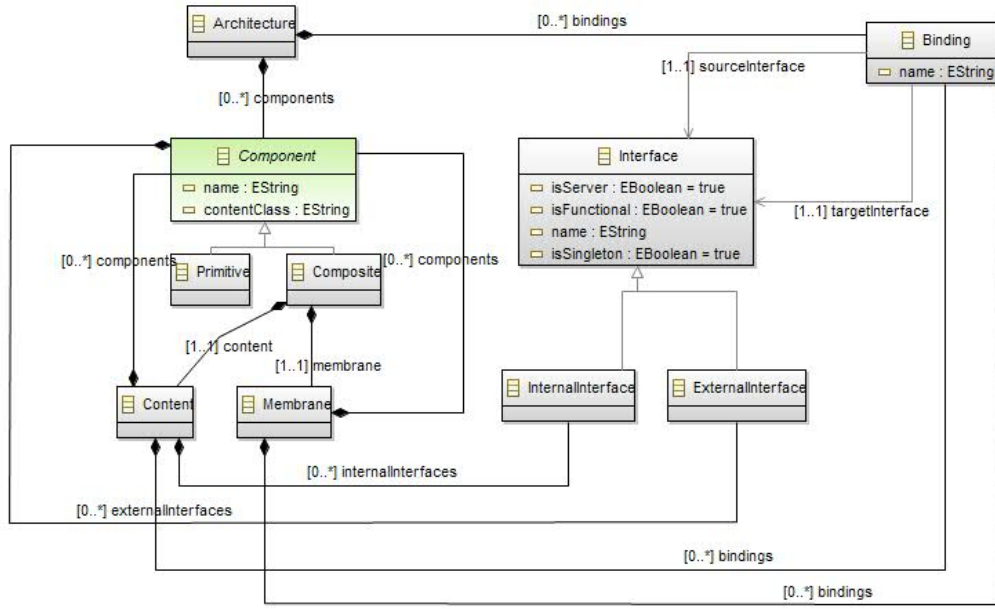


Figure 11: vce.ecore

name and Content class. Content class is a name of a class containing the code which implements the business logic of a component. Component has two ancestors: Primitive and Composite. A Primitive component has a reference to a UML State Machine, which describes its behavior. A UML State Machine is included in uml.ecore. A Composite component owns two mandatory elements: Content and Membrane. Both Content and Membrane can contain other components and bindings.

Another element of the vce.ecore metamodel is Interface. It has four parameters. IsServer defines if it is a server or a client interface. IsFunctional parameter defines if an interface corresponds to a functional or to a non-functional concern. In addition, an interface has a name. IsSingleton parameter defines if it is a singleton interface. An interface has a reference to a UML class in the uml.ecore. The list of the operations of a referenced UML class represents a list of methods of a VCE interface. The Interface has two ancestors: InternalInterface and ExternalInterface. The instances of the former one are included in the Contents. The instances of a latter one belong to the Components.

Finally, a Binding represents a connection between interfaces on a VCE Components diagram. It has a name and two references. One to a source interface and another one to a target interface.

The EMF editor for the GCM architecture semantic models was generated automatically from vce.ecore.

#### 4.3.2 Graphical editor for VCE Components diagrams implementation

The VCE Components diagrams graphical editor in VCE v.3 is based on vce.odesign file. In addition, it uses a number of Java classes for implementation of some specific features and diagrams validation. This section provides the description of vce.odesign file structure and a brief overview of the Java classes linked to it. The diagrams validation rules are defined in

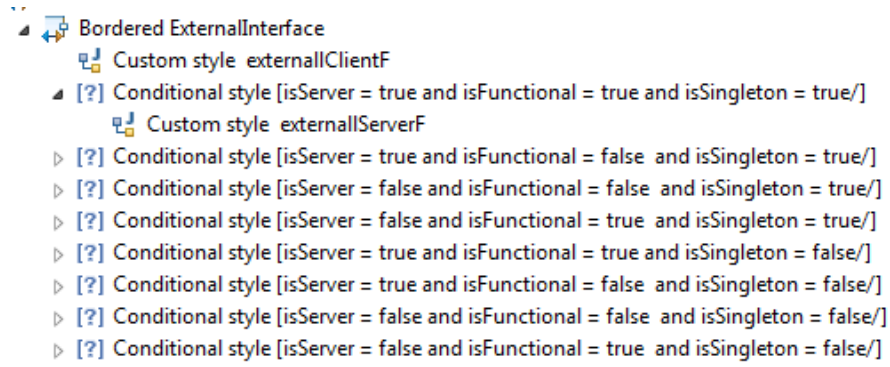


Figure 12: The conditional style specification in vce.odesign file

vce.odesign file but we describe them in a separated section.

Vce.odesign file contains the description of the graphical representation of the VCE Components diagram elements, the description of tools used in order to edit them and a set of rules for the diagrams validation.

According to vce.odesign description, primitive components are represented as containers. It might seem strange because according to vce.ecore primitive components do not include any other elements. However, we use containers but not simple nodes in order to display State Machines inside a primitive component. Composite components are represented as containers which include only two other elements: membrane and content. Membrane and content are represented as containers because they can contain other components and bindings.

Internal and External Interfaces are represented as bordered nodes that can belong to a content or a component correspondingly. Conditional style is used in order to display the interfaces. It means that the picture illustrating an interface changes at runtime depending on the parameters of an interface. For example, a blue cycle is used for a server functional singleton interface while a green cycle is used for a server non-functional singleton interface. In order to describe a conditional style in Obeo Designer we write Aceleo expressions which are evaluated at runtime. For each rule we define the corresponding style. If at some moment during the graphical editor execution an Aceleo expressions is evaluated to true, then the corresponding style is applied. Figure 12 illustrates the conditional styles description for the external interfaces

The Bindings are represented as the element-based edges.

A number of tools is specified for the edition of components, interfaces and bindings.

First, the instruments for the components should be described. There are tools for the primitive and composite components creation. When a composite component is created, its membrane and content are automatically generated too. The vce.odesign file has stubs for the content and membrane delete tools. According to their specification, nothing happens when a user tries to delete a membrane or a content. The stubs are needed because by default, if no delete tool is specified for a diagram element, then this element is deleted when a user presses on delete button. In our case, a membrane and a content are mandatory elements and they can be deleted only with the composite component containing them. Also, there are drag and drop tools for the components moving to a membrane, content or the the root layer of a diagram which is Architecture. Finally, there is a tool that allows to attach a UML State Machine to a primitive component. When a user right-clicks on a primitive component a menu is opened. The tool adds an Attach UML State Machine option to this menu. When a user chooses the

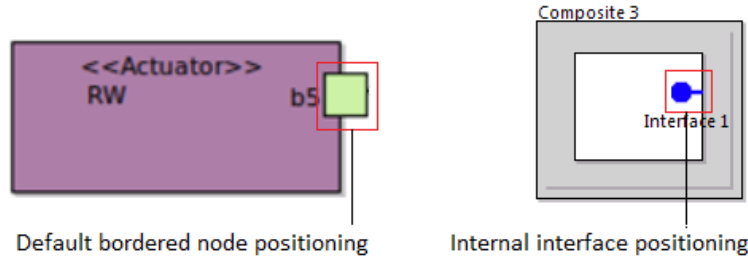


Figure 13: The positioning of internal interfaces

option, a list of available State Machines appears. The creation of such list is not a trivial issue and it is described in the section[REF].

Second, a number of tools is provided for the interfaces. The creation tools allow user to add interfaces with different parameters: singleton client and server interfaces, multicast and gathercast interfaces. The parameters can be changed after an interface is created. However, we provide several different tools in order to facilitate the creation. The same tools are used for the internal and external interfaces creation. If a user attaches an interface to a content, an instance of an internal interface is created. If a user attaches an interface to a component, then an instance of an external interface is created. Finally, there is a tool that allows to attach a UML class to a GCM interface. Its mechanism is similar to the one for a UML State Machine attachment.

Finally, there are tools for the bindings creation and reconnection, for the direct edit of bindings, components and interfaces names.

Vce.odesign file has references to Java classes. The classes are used for the implementation of specific features, elements custom styles and diagrams validation. Some of them are described in details in the following sections.

#### 4.3.3 Implementation of specific features in VCE v.3

Implementation of some features is not based on the standard mechanisms provided in Obeo Designer. Furthermore, sometimes we implement functionality which is claimed to be unfeasible. This section provides the overview of the most interesting specific features implementation.

##### Internal interfaces positioning

Recall, that the internal interfaces are represented as the bordered nodes in VCE v.3. By default, a bordered node is positioned outside its container. However, we want to position an internal interface inside a content. Figure 13 illustrated the difference between the standard and the required bordered nodes positions.

Obeo Designer allows to define a layout manager for a bordered node. We used this mechanism in order to implement the positioning of internal interfaces in the following way.

- We created a Java class *InternalPortStyleConfiguration*. It implements *fr.obeo.dsl.viewpoint.diagram.tools.api.graphical.edit.styles.BorderItemLocatorProvider* interface. It has *getBorderItemLocator()* method. The method takes a figure of an internal interface as an input and returns an instance of class *fr.obeo.dsl.viewpoint.diagram.ui.tools.api.figure.locator.DBorderItemLocator*. The returned instance locates the internal interface inside of its container.

- We created a Java class *InternalPortStyleProvider*. It implements *fr.obeo.dsl.viewpoint.diagram.tools.api.graphical.edit.styles.IStyleConfigurationProvider* interface. It has a function *createStyleConfiguration()* which creates and returns an instance of *InternalPortStyleConfiguration* class. It also has a function *provides()*. It returns true if the argument passed to it is an instance of *InternalInterface*.
- We registered *InternalPortStyleProvider* in the plugin.xml file by adding the following extension:

---

```
<extension
    point="fr.obeo.dsl.viewpoint.diagram.styleConfigurationProvider">
    <styleConfigurationProvider
        providerClass="vce.InternalPortStyleProvider">
    </styleConfigurationProvider>
</extension>
```

---

### Rotation of interfaces

A picture of an interface changes depending on the border to which an interface is attached. It is called rotation. Obeo Designer does not provide this functionality. However, it can be useful in many graphical editors. That is why when we decided to implement the rotation of interfaces, the description how to do it already existed on the Obeo Community forum. According to it, we implemented the interfaces rotation in the following way:

- We downloaded the source code of *fr.obeo.dsl.viewpoint.diagram.extension* provided by Pierre-Charles David, one of the Obeo Community members.
- We declared a number of custom styles in plugin.xml file. One for each combination of interface parameters values. More precisely, different styles are used for a functional client, a functional server, a functional multicast interfaces, etc. For each style we specified four images that will be used for an interface illustration: one for each border of a container. Different styles are required in order to use different pictures for the interfaces of different roles, cardinalities and concerns. An example of a custom style declaration for the external server singleton functional interface rotation is given below.

---

```
<extension
    point="fr.obeo.dsl.viewpoint.diagram.extension.rotatableBorderedNode">
    <image
        bottom="/vce.design/icons/serverInterface/externalServerBottomF.gif"
        id="externalServerF"
        left="/vce.design/icons/serverInterface/externalServerLeftF.gif"
        right="/vce.design/icons/serverInterface/externalServerRightF.gif"
        top="/vce.design/icons/serverInterface/externalServerTopF.gif">
    </image>
</extension>
```

---

- We specified the graphical representation of interfaces in *vce.odesign* file as conditional custom styles. For each combination of interfaces parameters a certain custom style is applied.

### Unique names generation

When a user creates a new component or a new interface, a unique name is computed and assigned to the created element. A user can change the name later.

In order to generate names we use the same approach as in `uml.odesign` for the unique UML classes names generation. For example, when a new primitive component is created, the following expression is computed and set as its name: *Primitive [eContainer().eContents().filter("Component").nSize()]/*. Primitive is a static part of expression while the remaining part is computed at runtime. As a result, it gives the number of components in the container where the new component is created. The same approach is used for the composite components, internal and external interfaces.

### Components names positioning

Containers are used for the components representation. By default, a name of a container is placed inside it. The current version of Obeo Designer does not provide any mechanism for the changing of a container name position. However, we managed to place the names of components outside. In order to achieve the goal we implemented the following actions:

- We created a class *MyLabelEditPart* which extends *fr.obeo.dsl.viewpoint.diagram.internal.edit.parts.DNodeNameEditPart* class. It has a function *refreshBounds()* which positions a name label outside a container.
- We created a class *MyContainerLabelEditPartProvider* which extends *org.eclipse.gmf.runtime.diagram.ui.services.editpart.AbstractEditPartProvider*. It has a function *getNodeEditPartClass()* with a single argument. If an argument passed to the function is a component, then it creates and returns an instance of *MyLabelEditPart*.
- We registered *MyContainerLabelEditPartProvider* in `plugin.xml` file.

### Layout manager for the composite components

One of the most challenging tasks was to implement layout for the composite components. Each composite component has two elements inside: a membrane and a content. A membrane should fill its container. Hence, when the size of a composite component is changed, the size of its membrane should be changed too. A content lies on the top of a membrane. Its size is not changed when the composite component is resized.

Recall, that we use containers for the composite components implementation. Obeo Designer does not provide any mechanisms for a custom layout manager definition of containers. Eventually, we managed to assign our own layout manager to the composite components in the following way:

- We created a class *MyCompartmentLayoutManager*. It extends *org.eclipse.gmf.runtime.diagram.ui.layout.FreeFormLayoutEx* class. It has a function *layout()* which arranges the contained content and membrane.
- We created a class *MyCompartmentEditPart*. It extends *fr.obeo.dsl.viewpoint.diagram.internal.edit.parts.DNode ContainerViewNodeContainerCompartment2EditPart* class. It has a function *getLayoutManager()* which returns an instance of *MyCompartmentLayoutManager* class.
- We created a class *MyCompartmentEditPartProvider*. It has a function *getNodeEditPartClass()*. If a component is passed to the function as an argument, then it returns an instance of *MyCompartmentEditPart*.

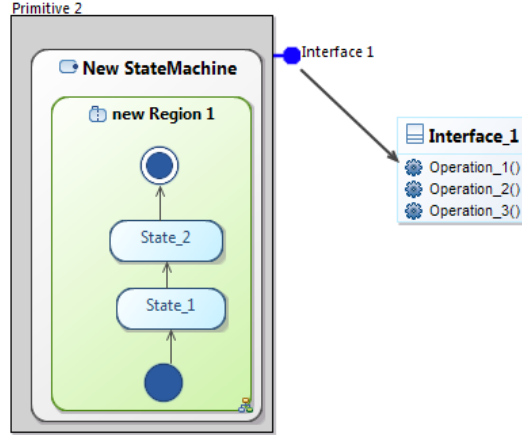


Figure 14: An example of a VCE Components Diagram displaying the referenced UML class and State Machine

- We registered *MyCompartmentEditPartProvider* in the plugin.xml file.

### Integration with UML diagrams

In order to display a UML State Machine inside a primitive component we implemented the following actions:

- Obeo Designer allows to Load resource in an .odesign file. Using this function one can link together several .odesign files. We loaded uml.odesign in vce.odesign file in order to use UML elements in VCE Components diagram editor.
- In vce.odesign we specified a new contained element State\_Machine\_Import in a primitive component. State\_Machine\_Import is a container. It completely reuses the functionality and all the contained element of a State Machine element from uml.odesign.

In order to display the UML classes linked to the GCM interfaces, we implemented the following actions:

- We created a new element Imported\_Class in vce.odesign file. It is an import of a UML class representation definition from um.odesign. Imported\_Class is a specification of the UML classes linked to GCM interfaces representation.
- We defined a new relation-based edge, which connects a GCM interface and the corresponding UML class.

Figure 14 Illustrates an example of a VCE Components diagram with the referenced UML State Machine and class.

### Creation of the available State Machines list

In order to attach a UML State Machine, a user right-clicks on a primitive component. Then, a menu appears. A user chooses Attach State Machine option from this menu. Finally, he selects a State Machine from a list of available State Machines.

In order to add the option to the menu, we created a standard Obeo Designer tool. However Obeo Designer does not provide any mechanism for the formation of the State Machines list. In

order to implement it, we created a Java class *UMLStateMachinesService* with a method *getAllUMLStateMachines()*. It gets a primitive component representation as an argument. Then it gets all the elements of the models from the project containing the primitive component. Finally, the function chooses UML classes from this set and returns them. *getAllUMLStateMachines()* is called when the tool discussed before is applied.

#### 4.3.4 Diagrams validation

A user can validate most of the consistency rules using VCE v.3. This section describes how the validation is implemented.

Obeo Designer provides a standard mechanism for the validation rules specification. First, a developer creates Validation section in his .odesign file. Second, he adds a Semantic Validation rule item to the section. Finally, for the semantic validation rule he specifies the following parameters:

- **Level.** Defines how critical the violation of the rule is. There are three available levels: ERROR, WARNING and INFORMATION.
- **Target class.** The rule validation is performed on all instances of this specified class.
- **Message.** The message is shown when the rule is violated. It can include both static and dynamic expressions (e.g. the values of elements parameters). The resulting message is computed at runtime.
- **The audit expression.** This expression is evaluated when the rule is checked. If the audit expression is evaluated to false the rule is violated. Acceleo and OCL languages can be used for the evaluated expressions. Also, Java methods can be called from the evaluated expressions.
- **Fix expression.** The expression is evaluated in order to fix the model in case the rule is violated. We do not use fix expressions in the current version of VCE.

Nine validation rules are specified in VCE v.3. Some of them are represented as OCL expressions, some of them call Java methods. In the remaining part of this section, we provide the examples constraints implementation using both methodologies.

##### An example of a constraint implementation based on OCL

According to one of the validation rules all the components in the same container must have unique names. The specification of this rule in vce.odesign file has the following parameters values:

- **Level:** ERROR;
- **Target class:** vce.Component;
- **Message:** The are several components with the name `{%name%}` in the same container. Here, expression `{%name%}` is evaluated at runtime. The value of the attribute name of the component on which the constraint is checked is substituted instead of it.
- **The audit expression:** `[not self.siblings(Component)->collect(name)->includes(self.name)]`. The audit expression uses OCL. In this case, *self* is the component on which the rule is checked. We will call it C. *self.siblings(Component)* gives all the components inside the container of C excluding C. *collect(name)* gives a set of all such components names. *includes(self.name)* is evaluated to true if the set of the names contains the name of C.



**An example of a constraint implementation using Java method call** According to one of the constraints, a binding must connect interfaces of compatible types. Its implementation is interesting because it checks the correctness of the VCE model integration with the UML model. Recall, that a UML class can be attached to a GCM interface. In order to check the constraint, the compatibility of the UML classes attached to each bindings end-interfaces must be checked. The specification of the considered rule has the following parameters values:

- **Level:** ERROR;
- **Target class:** vce.Binding;
- **Message:** The binding ;%name%i connects interfaces with incompatible methods.
- **The audit expression:** [self.isSubtypingCorrect()/]. *Self* is the binding on which the constraint is checked. *isSubtypingCorrect()* is a method of a Java class vce. services.BindingValidationServices. The class is referenced from the vce.odesign file. The code of the *isSubtypingCorrect()* function is given below.

---

```

public Boolean isSubtypingCorrect(Binding binding) {
    Interface srcGcmItf = binding.getSourceInterface();
    Interface targetGcmItf = binding.getTargetInterface();
    org.eclipse.uml2.uml.Class srcUmlItf =
        srcGcmItf.getReferencedInterface();
    org.eclipse.uml2.uml.Class targerUmlItf =
        targetGcmItf.getReferencedInterface();

    if(srcUmlItf == null && targerUmlItf == null)
        return true;
    if(srcUmlItf == null)
        return false;
    if(targerUmlItf == null)
        return false;

    EList<Operation> srcOperations = srcUmlItf.getAllOperations();
    EList<Operation> targetOperations = targerUmlItf.getAllOperations();
    ArrayList<Operation> usedOperations = new ArrayList();
    for(Operation op1 : srcOperations) {
        boolean existsCompatible = false;
        for(Operation op2 : targetOperations) {
            if(checkOperationsEquality(op1, op2) &&
                !usedOperations.contains(op2)) {
                usedOperations.add(op2);
                existsCompatible = true;
                break;
            }
        }
        if(!existsCompatible) {
            System.err.println("There is no compatible operation for "
                + op1);
            return false;
        }
    }
    return true;
}

```

---

The function gets a binding as an argument. It returns true if the binding connects GCM interfaces referencing to UML classes of compatible types or if the UML classes are not specified for both GCM interfaces. Otherwise, it returns false.

First, the function extracts the source and target GCM interfaces and the referenced UML classes. Second, it checks if both referenced UML classes are specified. If both of them are specified, it gets the lists of their methods. Finally, for each method of a source UML class it tries to find the corresponding method on the target class. If such operation is not found the function returns false.

To conclude, using the Obeo Designer technology we implemented a new version of VCE - VCE v.3. It has standard EMF editors for GCM architecture and UML semantic models. They allow to see the models in a form of a tree and edit them. We developed a graphical designer for VCE Components diagrams. It provides the graphical representation of the GCM architecture semantic models and a huge set of tools for their edition. We integrated the existing graphical editor for UML diagrams into VCE v.3. We implemented the functionality which has not been released in any version of VCE - integration of GCM architecture and UML semantic models. Now, a UML State Machine can be attached to a primitive component and a UML class can be attached to a GCM interface. The attached UML elements can be displayed on the VCE Components diagrams. VCE v.3 has a module for the developed GCM architecture semantic models validation.

## 5 Related work

In this section we provide the related work. First, we present a brief overview of the Fractal and Grid Component Model specifications. Second, we present two works on formal models for Fractal and GCM based applications. Third, we present the feasible technologies for the VCE v.3 implementation.

The latest version of the Fractal Component Model specification was released in 2004 by E. Bruneton, T. Coupaye and J.B. Stefani. It can be found here: [14]. The specification presents, first, the general idea of Fractal model. Second, it gives the description of the basic elements of the model: components, bindings and interfaces. A lot of attention is paid to the control properties of the interfaces. Third, it specifies a framework for the components creation which is based on the factories. Then, the properties of the interfaces are specified in details. Finally an example of a concrete model is provided.

The latest version of the Grid Component Model specification can be found in [22]. Its structure is very similar to the one of the Fractal specification.

The formal specification of the Fractal Component Model was developed by Philippe Merle and Jean-Bernard Stefani in Alloy [21]. Alloy is a simple specification language based on the first-order relation logic. The proposed formal model covers all the elements of Fractal defined in [14]. First, the authors introduce a structure for the components representation called *kell*, a structure for the subcomponents representation - *subcell*, a structure abstracting the notion of interfaces called *gate*. Each *kell* has an identifier, a set of subcomponents and a set of gates. The notion of inheritance in Alloy is used for the representation of the Client and Server interfaces. They are defined as the successors of a gate structure. Second, the authors define a structure *Signal* specifying the signature of a method that can be invoked on a gate. Third, the *Binder* structure establishes communication between two gates. Finally, the authors introduce a set of predicates expressed in Alloy. The predicates define the static semantics validity rules. However, even if the Fractal specification is fully covered by the proposed model, it is not appropriate for the Grid Component Model, because it does not define the componentized membrane and interceptors. The non-functional interfaces are defined in the proposed formal model but not the components performing the control acridities.

A formal specification of the Grid Component Model using the COQ theorem prover is provided in [17]. Here, the authors define a structure *Interface* with a set of characteristics, a structure *Component* and a structure *Binding*. Among the other properties, each component has a control level which defines if it can be internally reconfigured. The authors define well-formness of all the GCM core elements: components and bindings. However, the presented formal model does consider the fact that a composite component is divided into a content and a membrane. It does not define the interceptors. Also, the authors present a case-study. The main achievement of this work is the development of an operating language for the GCM-based architecture reconfiguration which allow to construct the distributed software correctly.

As mentionned in section 4, two approaches were considered by our team as an alternative to the Obeo Designer technology for the VCE v.3 implementation. Both of them were based on the idea of a VCE Components diagrams editor integration in Eclipse Papyrus [3]. Eclipse Papyrus is an Eclipse plugin providing a set of UML diagram editors.

The first approach tested by OASIS team was based on the UML profile [8] mechanism. The results of the experiment are presented in [15]. The UML profiles are supported in Eclipse Papyrus. They enable a user create his own semantics elements and their graphical representations and integrate them with UML models. The idea was to implement the GCM architecture elements as the UML profiles and integrate them with UML classes and State Machines imple-

mented in Papyrus. However, at some point it was observed that the rotation of the interfaces could not be implemented and, hence, the tested approach was not suitable for the VCE v.3 implementation.

The second idea was to create a VCE Components Diagram editor using Graphical Modeling Framework technology [5] and to integrate it into Eclipse Papyrus. As a result of the experiment, a simple prototype of a VCE Components diagrams editor was created and integrated in Eclipse Papyrus. It had tools for the primitive and composite components, interfaces and bindings creation. The interfaces were rotatable. However it was observed, that the development was extremely time-consuming and Papyrus project is not very well-documented and unstable.

## 6 Conclusions and future work

This work was focused on the modeling aspects of the distributed software development based on the Grid Component Model. The two main objectives of the work were achieved.

First, we extended the existing formal model of GCM-based architecture static semantics with respect to the non-functional aspect. The updated formal model is compatible with the current version of Architecture Description Language. It describes the membrane of a composite components, defines the notion of the interceptors and defines the functionality property of the interfaces. We gathered consistency constraints for the GCM-based architecture static semantics validation from the existing formal model, the ADL specification and the VCE v.2 graphical editor in a new coherent set of validation rules.

Second, we created a new version of VCE graphical editor called VCE v.3. It is based on Obeo Designer technology. VCE v.3 is distributed as a set of Obeo Designer plugins. It combines a graphical editor for the GCM-based applications architecture and a set of graphical editors for UML diagrams. VCE v.3 is the first tool which allows integration of GCM and UML models. A user can define a UML class with a set of methods and attach it to a GCM interface. It allows to validate the typing constraint for the binding connecting two interfaces. Also, a UML State Machine can be created and attached to a primitive component. It is a significant functionality required for the future behavior specification generation. The static semantics consistency rules validation was implemented in VCE v.3. The OASIS team is working on the ADL description generation from VCE v.3.

However, even if the goals of the work were achieved there is still a number of open questions and tasks remaining and we are going to continue working on them. First, for the formal model we still discuss the feasible functionalities of the interceptors. Also, the compatibility of the parameters values of the bounded interfaces is still discussed. Second, we want VCE v.3 to be a free software. We do not know whether the announced open-source version Sirius will cover all the functionalities of Obeo Designer, we are waiting for its release to test it. In the waiting, our users have to install Obeo Designer (evaluation, academic, or commercial version) on each computer where we want to execute VCE v.3. Third, it is still discussed if UML State Machines or UML Activity diagrams should be used for the primitive components behavior specification. In this work we integrated an editor for GCM-based static architecture with the one for UML State Machine diagrams. We suppose that the integration process should be similar for the UML Activity diagrams but this assumption has not been tested yet. Fourth, there is still a number of features which are not implemented in VCE v.3. They are not essential features but they can significantly facilitate the GCM-based applications modeling. For example, we would like to implement the copy-paste with layout and layout policies.

To sum up, in the future we would like to solve the mentioned open issues. Also, we are going to work on the other aspects of the GCM-based applications modeling which were not considered in this work. For example, such a huge issue as the consistency validation between GCM-based application behavior and architecture specification. We would like to create a module which would produce the behavioral specification in a form of pNets from the definition of architecture and primitive components' behavior. We would like to verify the produced pNets and provide a user with a description of the detected errors.

## References

- [1] Accelleo - transforming models into code. <http://www.eclipse.org/acceleo/>.
- [2] Eclipse modeling framework (EMF) homepage. <http://www.eclipse.org/modeling/emf/>.
- [3] Eclipse papyrus homepage. <http://www.eclipse.org/papyrus/>.
- [4] Eclipse project. <http://www.eclipse.org/>.
- [5] Graphical modeling framework (GMF) homepage. <http://www.eclipse.org/modeling/gmf/>.
- [6] Obeo designer: Methods and tools for architects. <http://www.obeodesigner.com/>.
- [7] Obeo designer starter tutorial. <http://www.eclipse.org/acceleo/>.
- [8] An open-source UML diagrams editor based on obeo designer. <https://github.com/ObeoNetwork/UML-Modeling/>.
- [9] Topcased: The open-spurce toolkit for critical systems. <http://www.topcased.org/>.
- [10] Uml 2.0 superstructure specification. Technical report, Object Management Group (OMG), August 2005.
- [11] S. Ahumada, L. Apvrille, T. Barros, A. Cansado, E. Madelaine, and E. Salageanu. Specifying Fractal and GCM Components With UML. In *proc. of the XXVI International Conference of the Chilean Computer Science Society (SCCC'07)*, Iquique, Chile, November 2007. IEEE.
- [12] Rabéa Ameur-Boulifa, Ludovic Henrio, Eric Madelaine, and Alexandra Savu. Behavioural Semantics for Asynchronous Components. Rapport de recherche RR-8167, INRIA, December 2012.
- [13] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. Technical Report RR-5591, INRIA, June 2005.
- [14] J.B. Stefani E. Bruneton, T. Coupaye. The fractal component model, February 2004.
- [15] Nicolas Le Halper. Développement d'éditeurs graphiques pour la plateforme Eclipse "Papyrus". Technical report, Université de La Rochelle, May 2012.
- [16] Oleksandra Kulankhina. Prototyping of a component architecture editor in the papyrus environment. Technical report, PFE Report, Université de Nice Sophia Antipolis, March 2013.
- [17] Nuno Gaspar Ludovic Henrio Eric Madelaine. Bringing coq into the world of gcm distributed applications. *accepted to Springer Science+Business Media New York*, 2013.
- [18] Paul Naoumenko. An extension for ADL definition of GCM components, taking into account the full specification of the membrane. Technical Report RR-7001, INRIA, 2009.
- [19] Paul Naoumenko. *Designing Non-functional Aspects With Components*. PhD thesis, University of Nice-Sophia Antipolis, July 2010.

- [20] Object Management Group. OCL 2.2 Specification. <http://www.omg.org/spec/OCL/2.2>, 2010.
- [21] Jean-Bernard Stefani Philippe Merle. A formal specification of the fractal component model in alloy. Technical report, Centre de recherche INRIA Grenoble Rhne-Alpes, November 2008.
- [22] ETSI TC-GRID. ETSI TS 102 829: Grid; grid component model; part 3: Gcm fractal architecture description language (adl). Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2009. standard.
- [23] OASIS Team. The VERCORS platform: Verification of models for distributed communicating components, with safety and security, 2010.

# Appendices

## A VCE v.3 User guide

# VCE v.3 Tutorial

RESEARCH CENTRE INRIA SOPHIA ANTIPOLIS MEDITERRANE  
OASIS TEAM

Author: Oleksandra Kulankhina  
oleksandra.kulankhina@inria.fr

Version: 0.1

July 2013



## 1. Introduction

The guide explains the basic functionality of VerCors v.3. It provides the instructions for creation of VCE models, VCE Components diagrams, UML Models, UML Class and State Machine diagrams and integration of a VCE models with UML models.

VCE v.3 is based on Obeo Designer technology and possess the functionality of the standard editors created with Obeo Designer. Hence, it is strongly recommended to read some of the Obeo Designer documentation in addition to this tutorial. In particular:

- Getting started for End-users:  
[http://docs.obeonetwork.com/obeodesigner/6.1/Getting\\_Started\\_User.html](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:

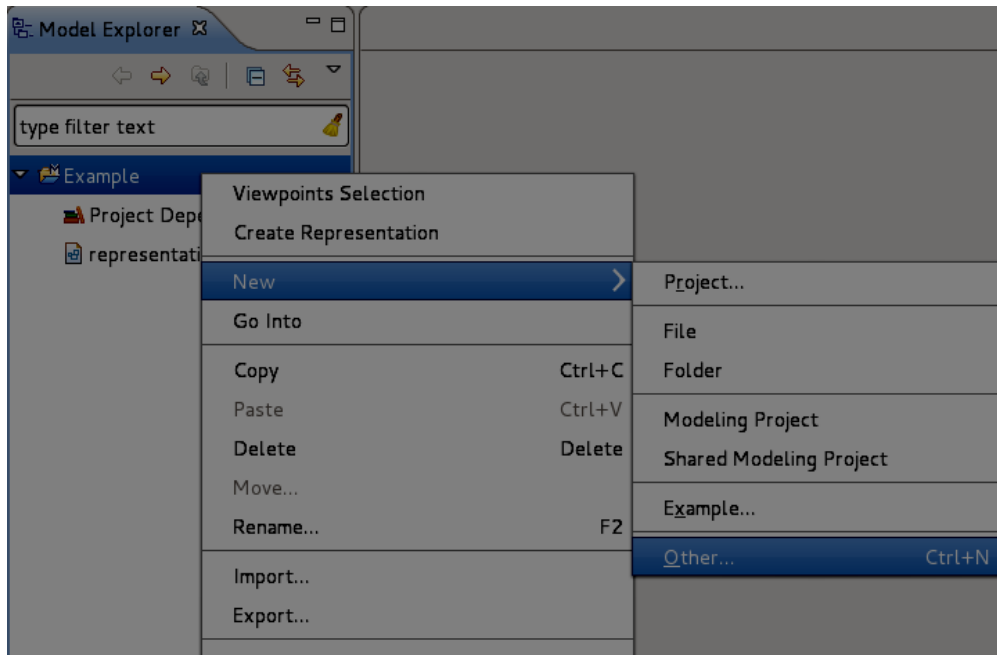
<http://www.obeonetwork.com/group/obeo-designer/page/obeo-designer-reference-document-6-1>

## 2. Overview

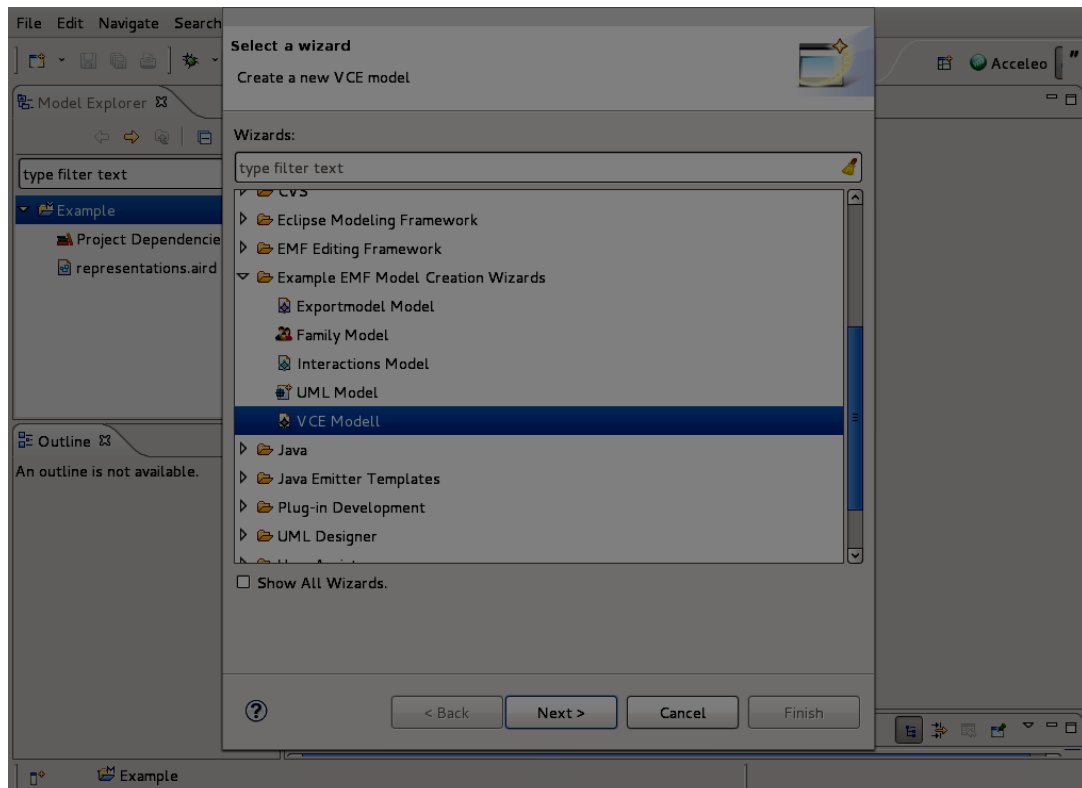
All the models and diagrams are stored in a **Modeling Project**. A Modeling project can contain **VCE and/or UML models**. It can also contain **VCE Components diagrams and UML diagrams** which illustrate the elements of the models. A VCE model can have links to UML models. In particular, a GCM interface can refer to a UML class. A GCM primitive component can refer to a UML State Machine which describes its behavior. The referred UML classes and state machines can be illustrated not only on UML diagrams but also on VCE Components diagrams.



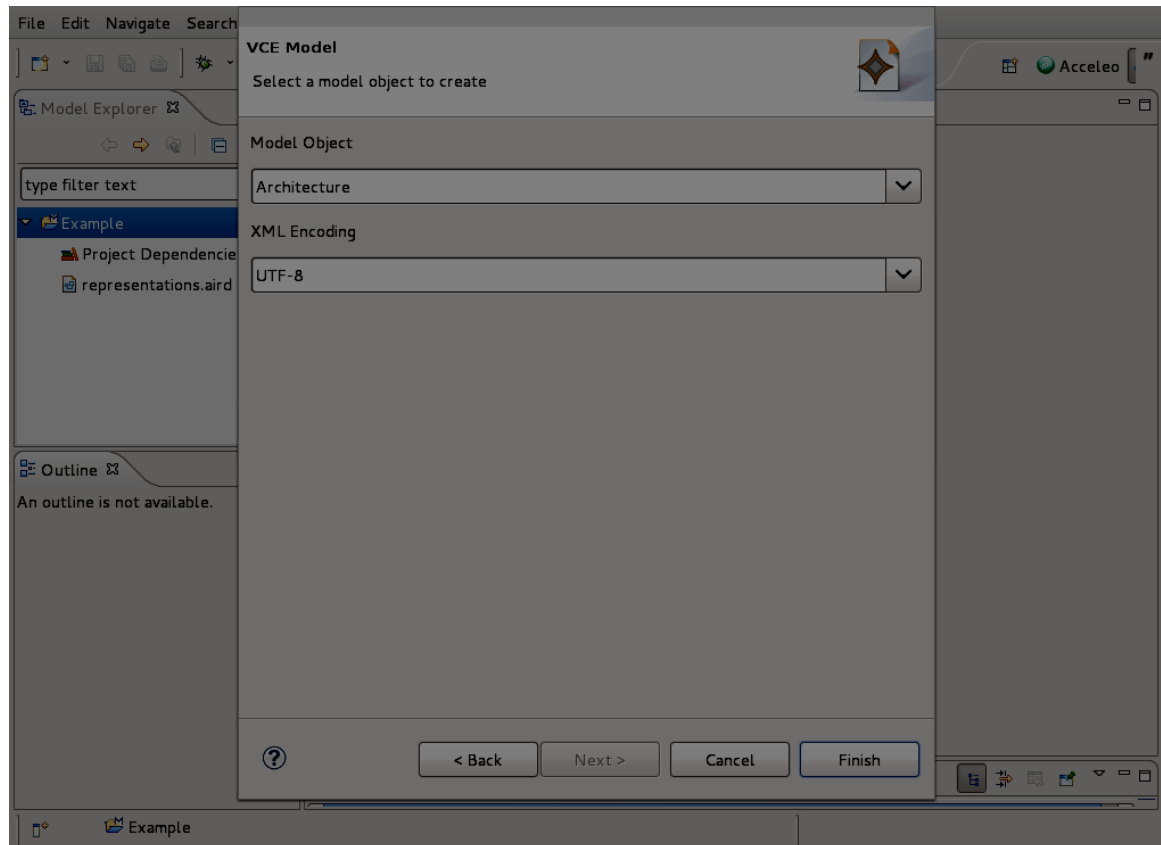
- Create a new VCE model. Right-click on the created modeling project and select “New->Other...”



- Select “**Example EMF Model Creation Wizard->VCE Model**” and press on “Next”



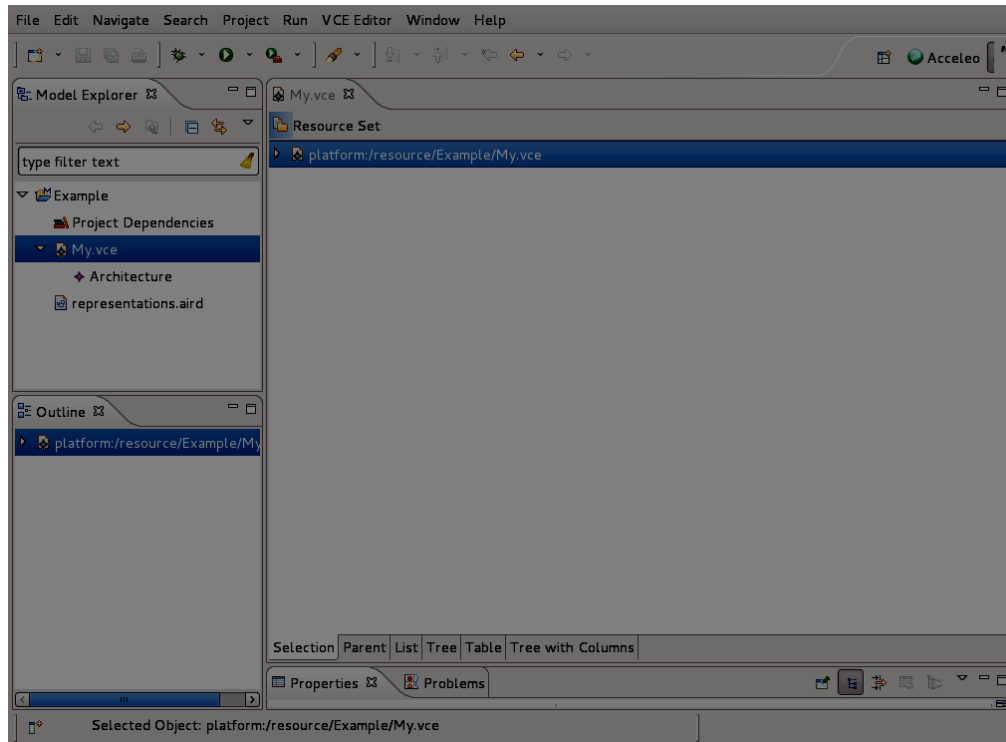
- Give a name to the model and press on “**Next**”. The name must be suffixed by “.vce”. We will call it “My.vce”.
- Set the Model Object to **Architecture** and press on “**Finish**”.



As a result, a new VCE Model should be created.

You can use a standard EMF Model editor in order to edit a VCE Model.

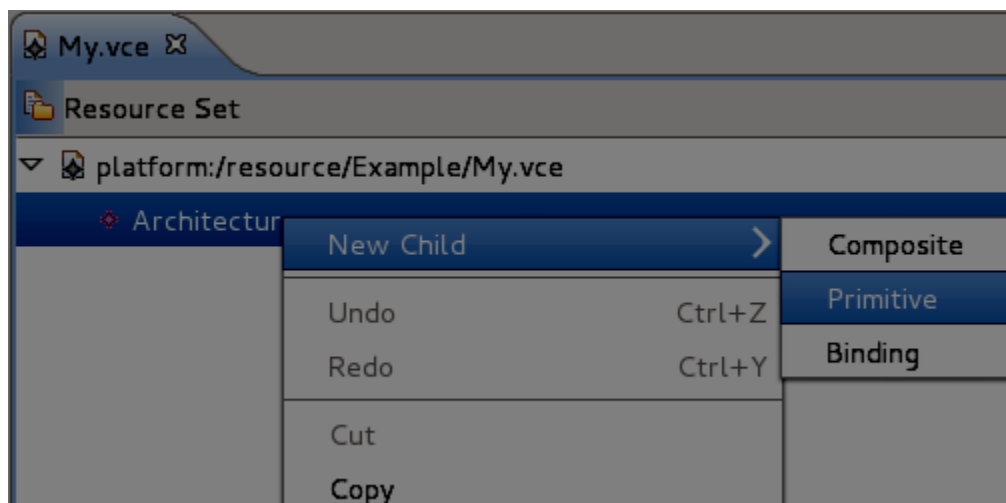
You can create as many VCE Models in the same project as you want but they all should have different names.



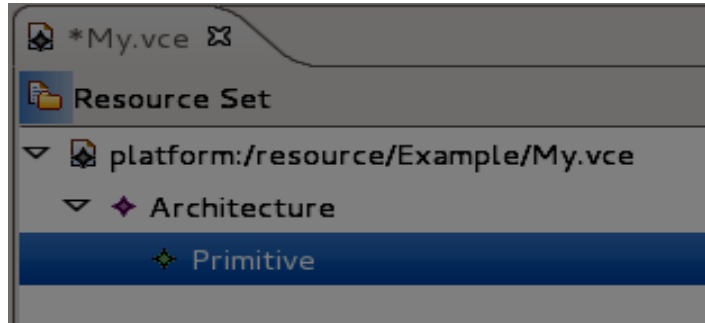
## 5. Edit model with the default editor

You can use a standard EMF Model editor in order to edit a VCE Model.

- Open the file with the created VCE Model using the default editor if it is not opened yet.
- Create a new primitive component in the Architecture. Right-click on the Architecture and select “**New child -> Primitive**”.



You should get the following result:

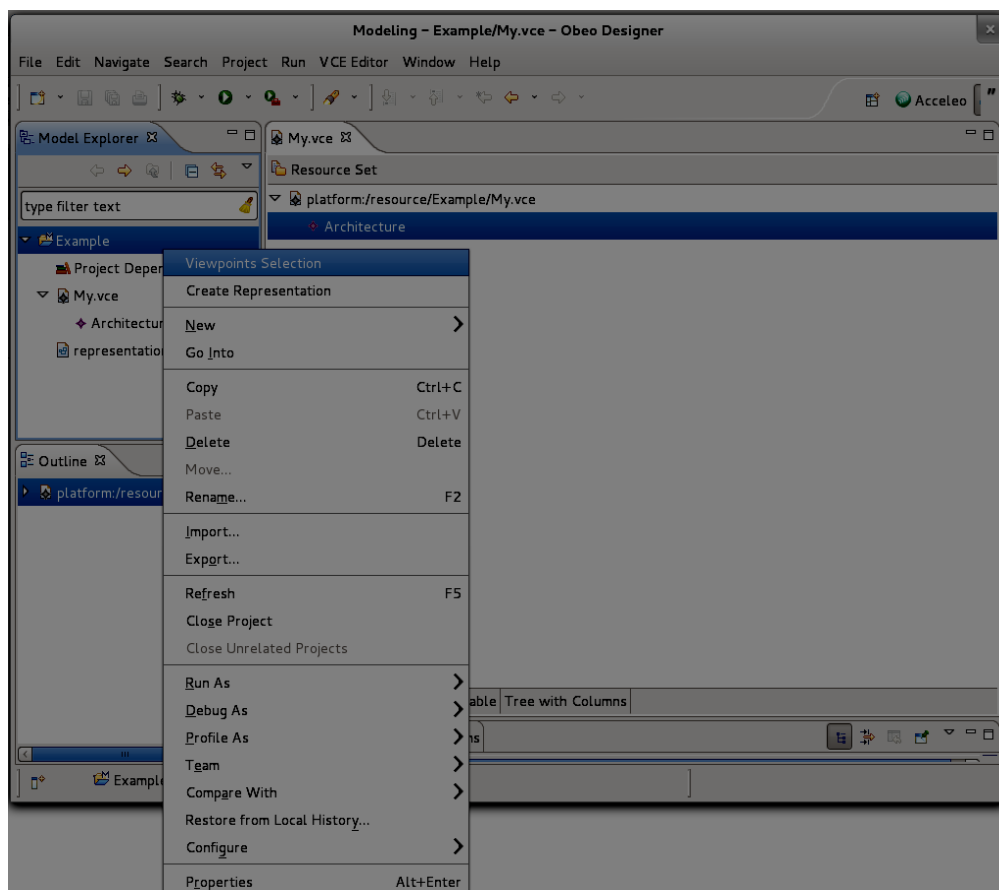


You can also create composite, primitive components and bindings inside the Architectures, Contents and Membranes.

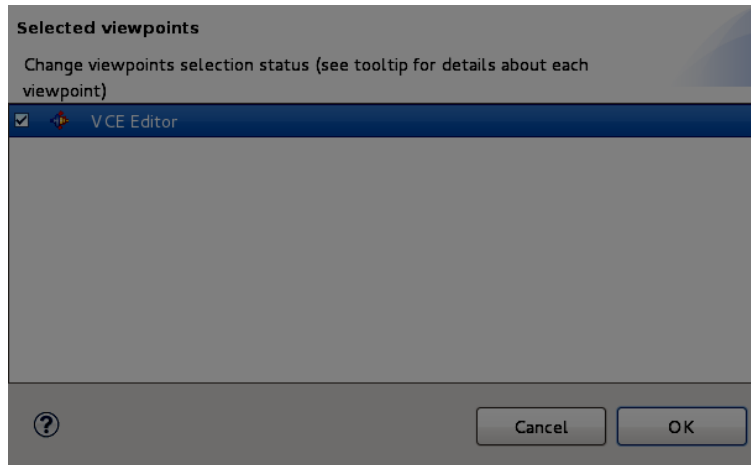
## 6. VCE Components diagram creation

The section describes how to create a diagram for a VCE Model.

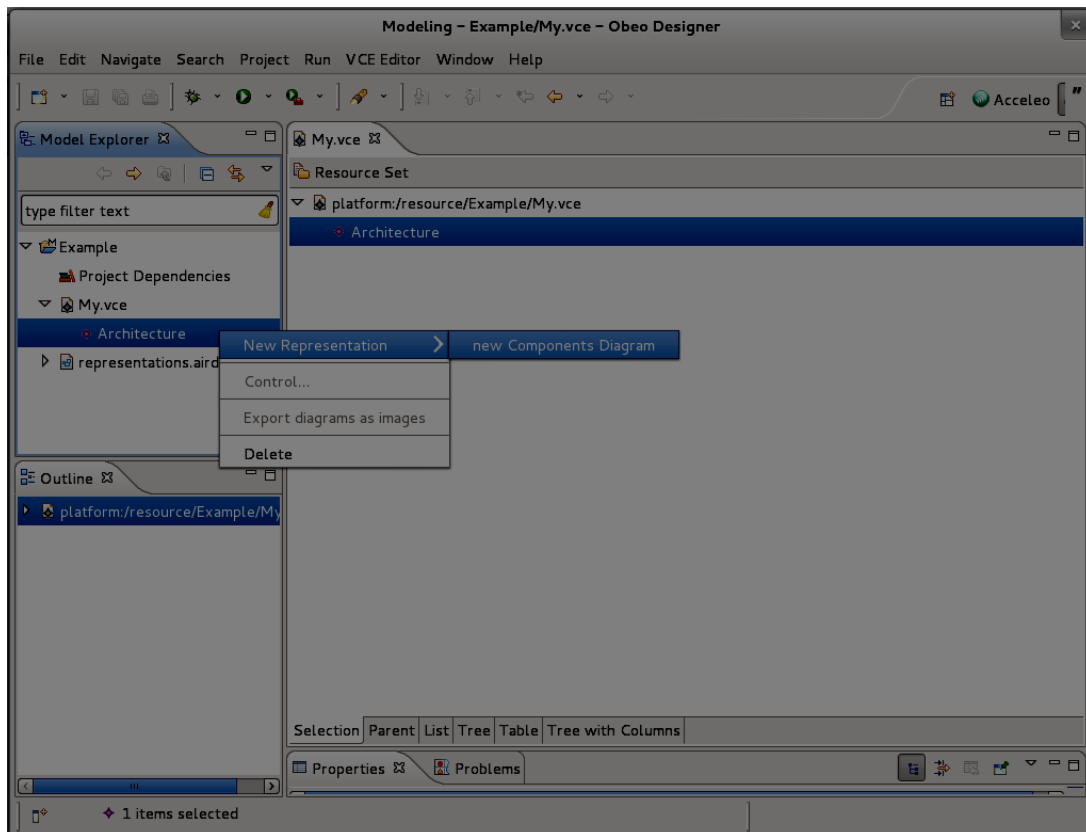
- Right-click on the Modeling project containing the VCE Model and select “**Viewpoint selection**” option.



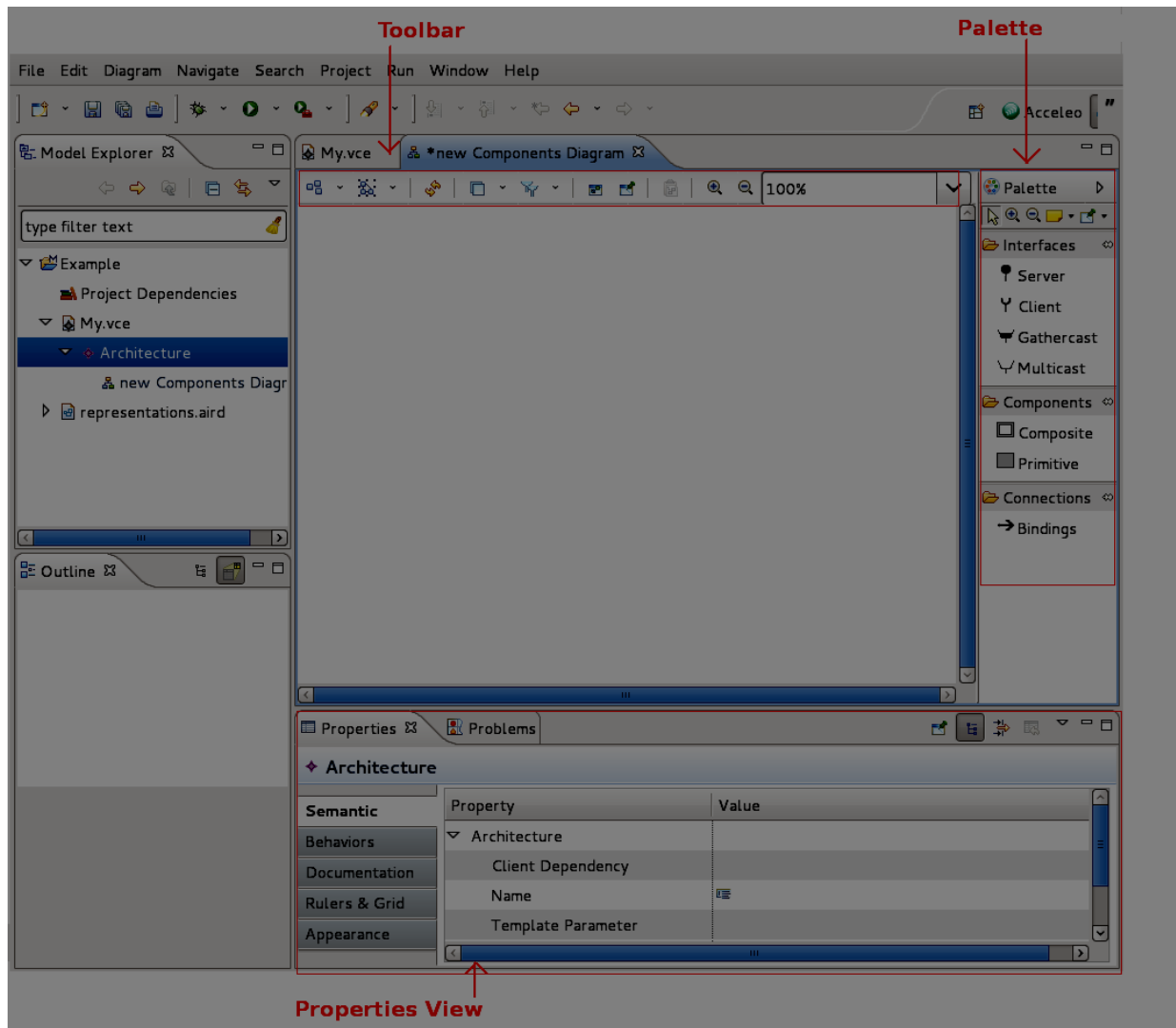
- Select “**VCE Editor**” in the opened window and press on “OK”.



- In the Model Explorer right-click on the Architecture of your VCE model and choose **“New representation -> new Components Diagram”**.



- Enter the name of the diagram and press on “OK”.
- A new diagram should be created and opened. 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.



## 7. UML Model creation

The section describes how to create a new UML Model. The process is very similar to the one for a VCE Model creation.

- Right-click on the Modeling project and choose “**New->Other...**”
- Select “**Example EMF Model Creation Wizard->UML Model**” and press on “**Next**”
- Give a name to the model and click “Next”. The name must be suffixed by “.uml”. We will call it “My.uml”.
- Set the Model Object to **Model** and press on “**Finish**”.

As a result, a new UML Model should be created.



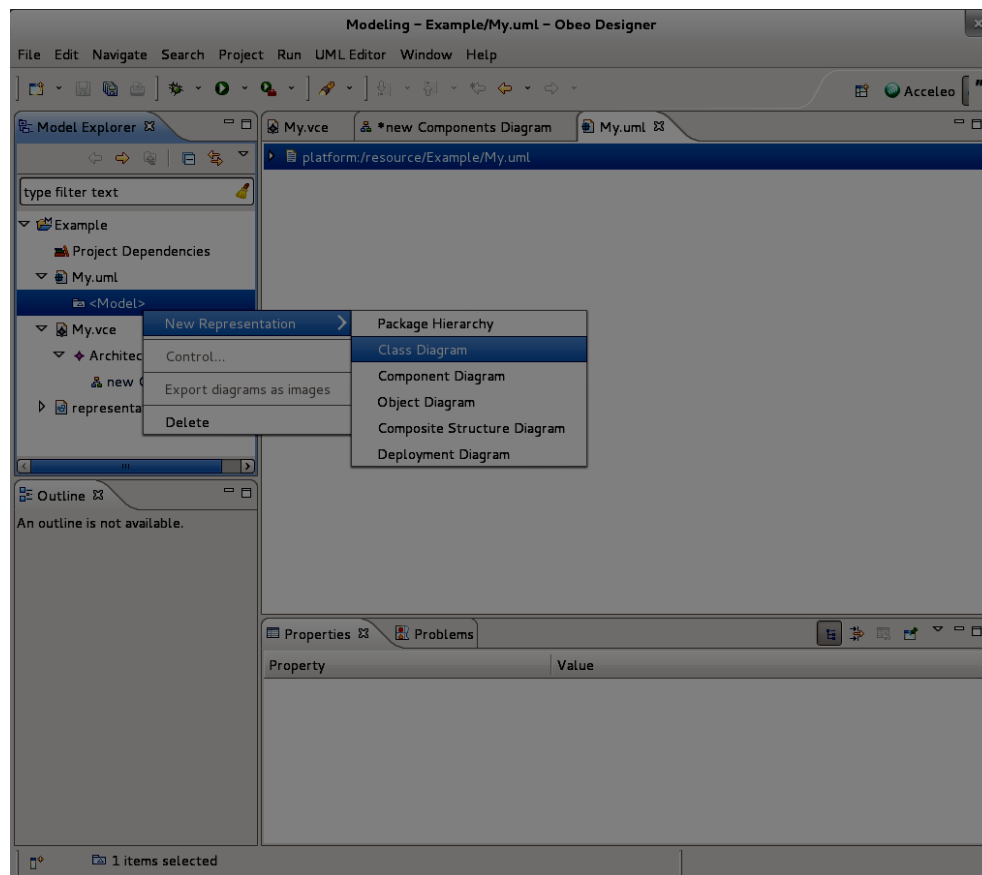
You can use a standard EMF Model editor in order to edit a UML Model.

You can create as many UML Models in the same project as you want but they all should have different names.

## 8. UML Class Diagram creation

The section describes how to create a diagram for a UML Model. The process is very similar to the one for a VCE Components diagram creation.

- Right-click on the Modeling project containing the VCE Model and choose “**Viewpoint selection**”.
- At this point “VCE Editor” option should be already selected. If it is not selected then you should choose this option. Select also “**UML Structural Modeling**” and press on “OK”.
- In the Model explorer right-click on the UML Model and select “**New representation -> Class Diagram**”.



- Enter the name of the diagram and press on “OK”.
- A new diagram should be created and opened. You can create new classes using tools on a

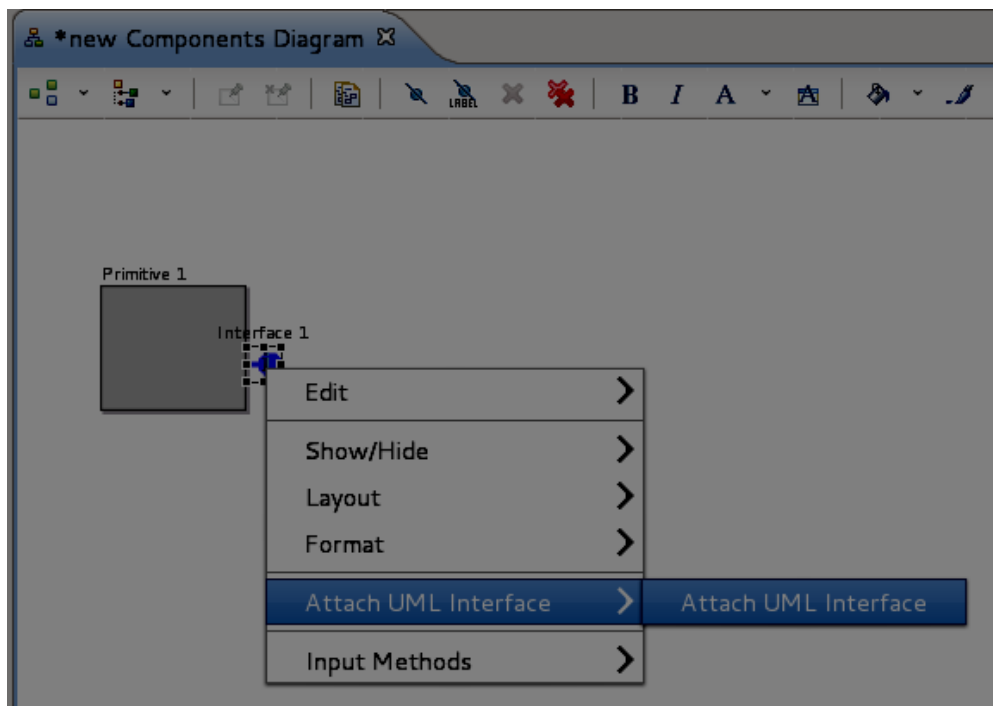
Palette on the right.

## 9. Attach a UML class to a GCM interface

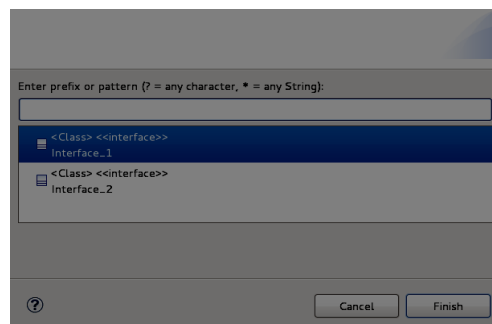
If you have a VCE Model with a GCM interface and a UML model with a UML class in the same project, then you can attach a UML class to a GCM interface. You can also display the corresponding UML class on a VCE Components Diagram. This section explains how to do it.

The precondition required in order to display a UML class on a VCE Components Diagram is that the “UML Structural Modeling” and “VCE Editor” viewpoints must be selected.

There are two ways to attach a UML class to a GCM interface.

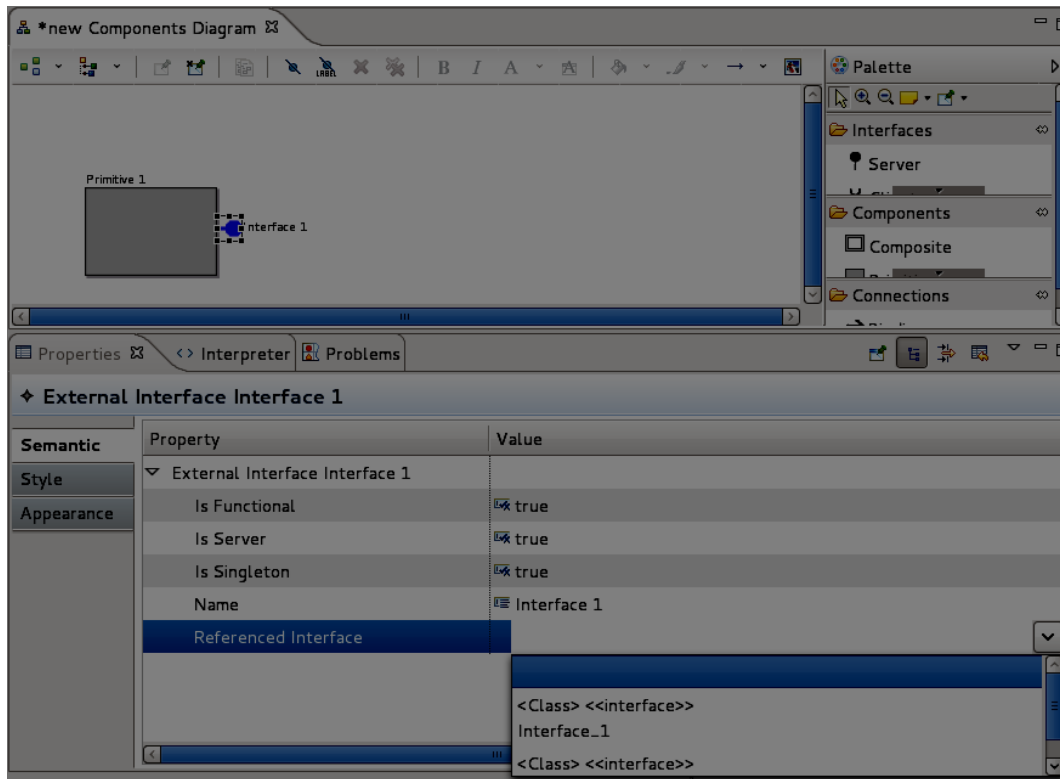


First, you can right-click on the GCM interface on your VCE Components diagram and select “**Attach UML Interface**” option.

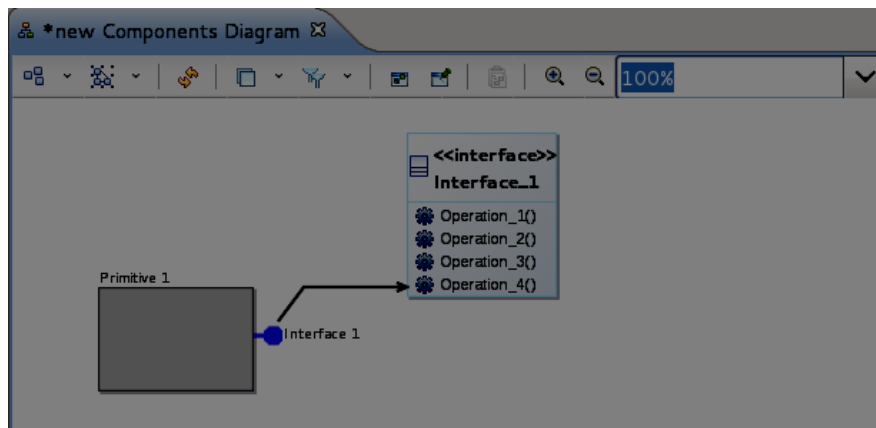


Then, you need to choose the required UML class from the list and press on “Finish”.

The second way is to click on the GCM interface on your VCE Components diagram and in its Properties view set the “**Referenced interface**” property to the corresponding UML class.

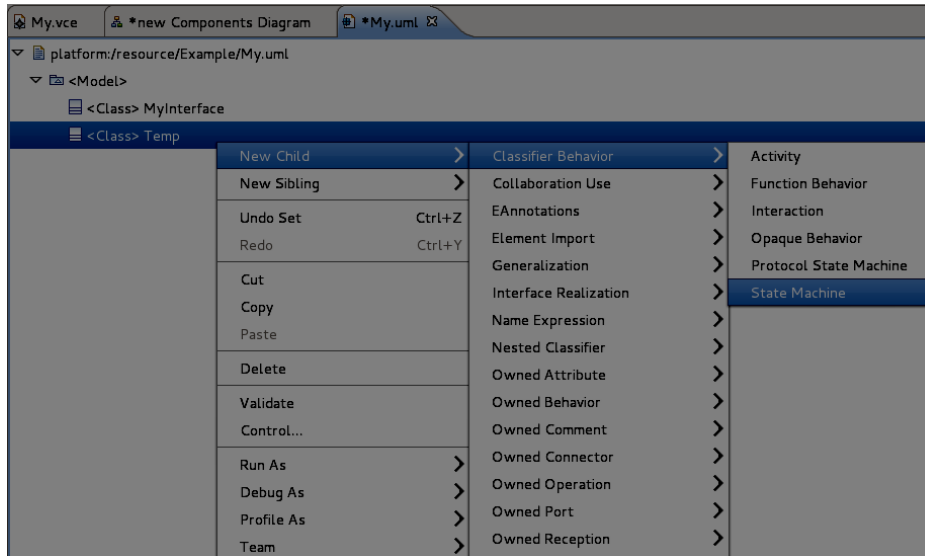


An example of a UML class attached to a GCM interface on a VCE Components Diagram is illustrated below.

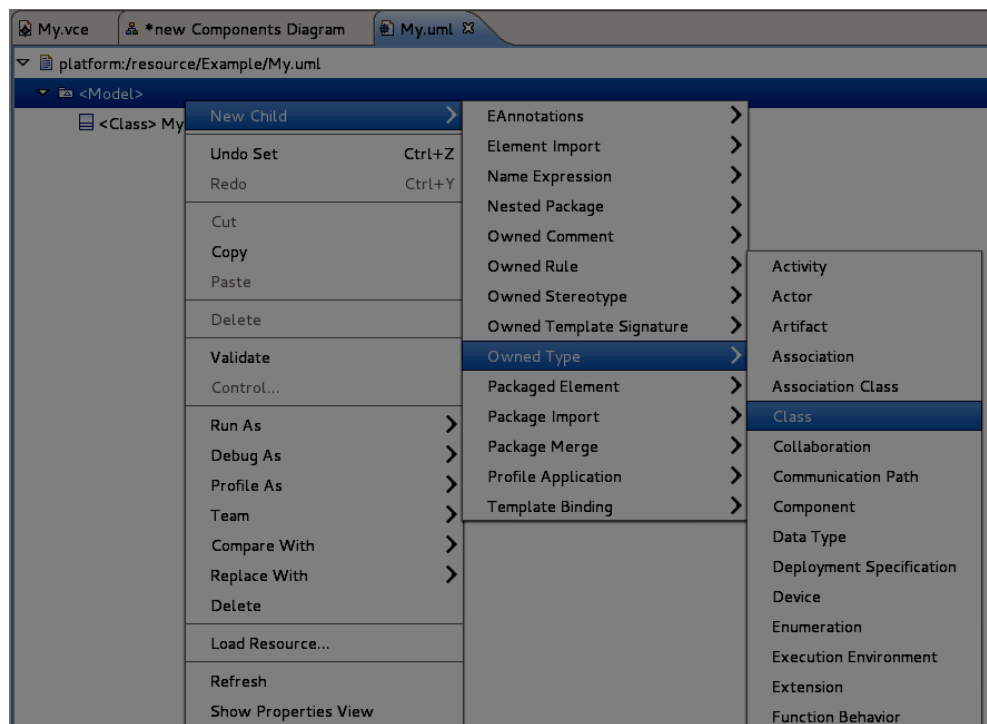


## 10.UML State Machine creation

The section describes how to create a UML State Machine that presents the behavior of a primitive component. **Note**, that a UML State Machine must be attached to a UML class. A UML State Machine cannot be created without a UML class even if you do not need the letter.



- Go to your UML Model file (.uml file).
- If you do not have a class to which you can attach a State Machine, then create it. Right-click on the root element of the UML Model which is <Model> and select “**New child -> Owned Type -> Class**”.



- Right-click on the class to which you want to attach a State Machine and select “**New child -> Classifier Behavior -> State Machine**”.
- Set the name of the State Machine in the properties view.
- Set the **isReentrant** property of the State Machine to false.

- Each State Machine must have at least one Region. In order to create it right-click on the State Machine and select “**New Child -> Region -> Region**”.
- Give a name to the Region in the properties view.

## 11. State Machine Diagram creation

The section describes how to create a diagram for a UML State Machine. The process is very similar to a UML Class diagram creation.

- Right-click on the Modeling project containing the VCE Model and select “**Viewpoint selection**”.
- At this point “VCE Editor” and “UML Structural Modeling” option should be already selected. If “VCE Editor” option is not chosen, select it. Select also “**UML Behavioral Modeling**” and press on “OK”.
- In the Model explorer right-click on the UML State Machine and select “**New representation -> State Machine Diagram**”.

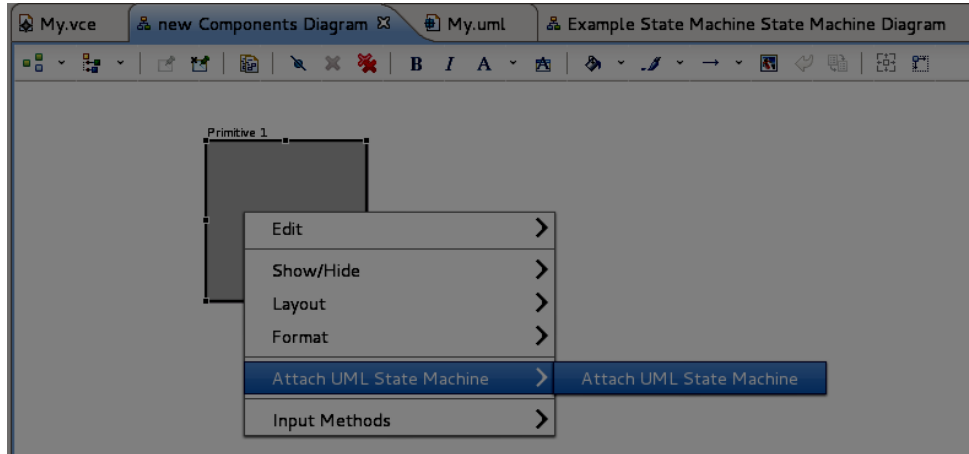
## 12. Attach a UML State Machine to a GCM Primitive Component

If you have a VCE Model with a GCM Primitive Component and a UML model with a UML State Machine in the same project, then you can attach a UML State Machine to a GCM Primitive Component. You can also display the former on a VCE Components Diagram. This section explains how to do it.

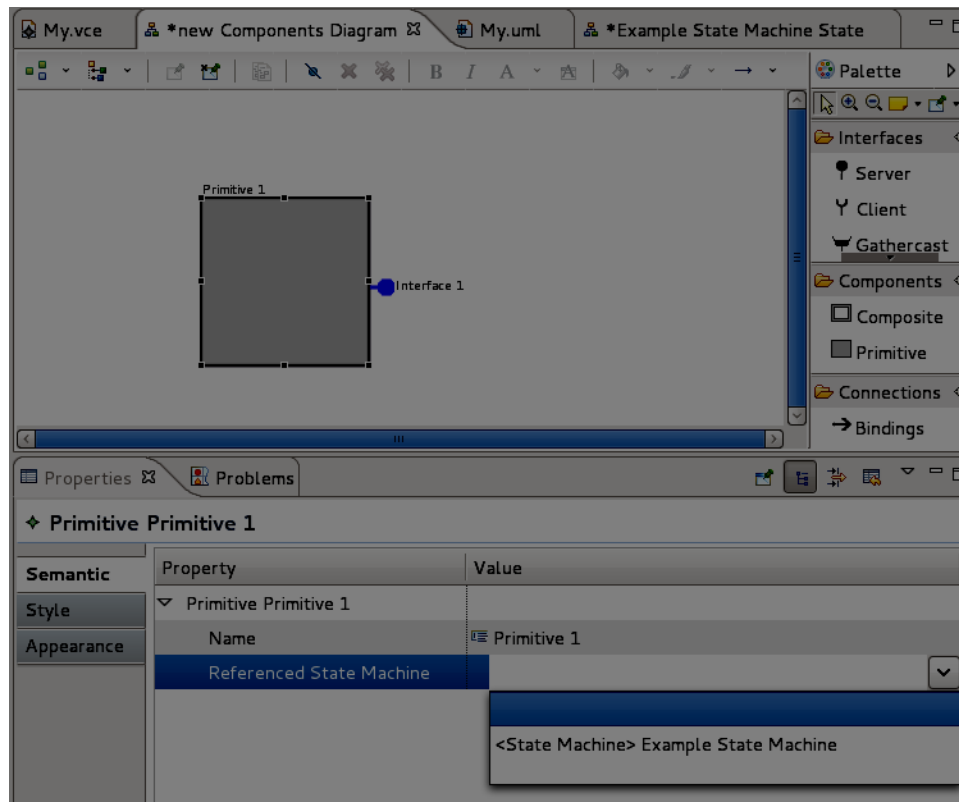
The precondition required in order to display a UML Class on a VCE Components Diagram is that the “UML Behavioral Modeling” and “VCE Editor” viewpoints must be selected.

There are two ways to attach a UML State Machine to a GCM Primitive Component.

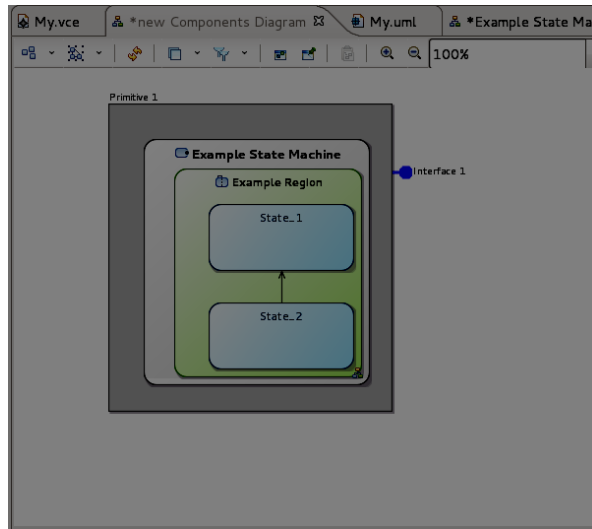
First, you can right-click on the GCM Primitive Component on your VCE Components diagram and select “**Attach UML State Machine**” option. Then, you need to choose the required UML State Machine from the list and press on “Finish”. The UML State Machine should appear on the VCE Components Diagram.



The second way is to specify the property “Referenced State Machine” of a GCM Primitive Component in its Properties View.



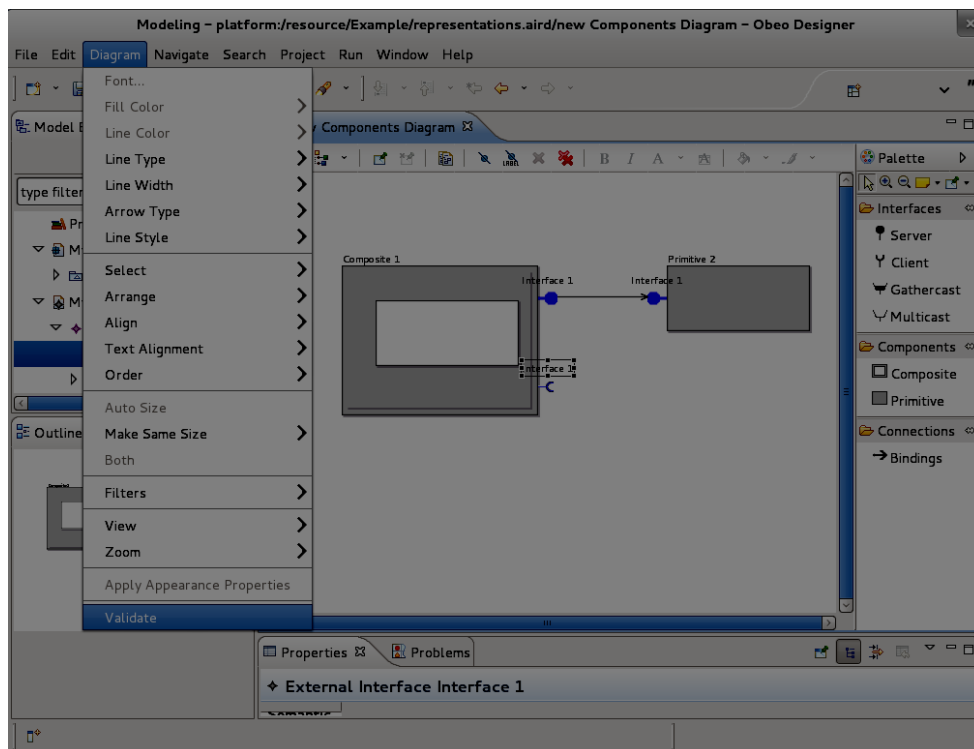
An example of a UML State Machine attached to a GCM Primitive Component is given below:



The same state machine attached to two different primitive components in the same VCE Model will not be displayed properly on a VCE Components diagram.

### 13. Diagrams Validation

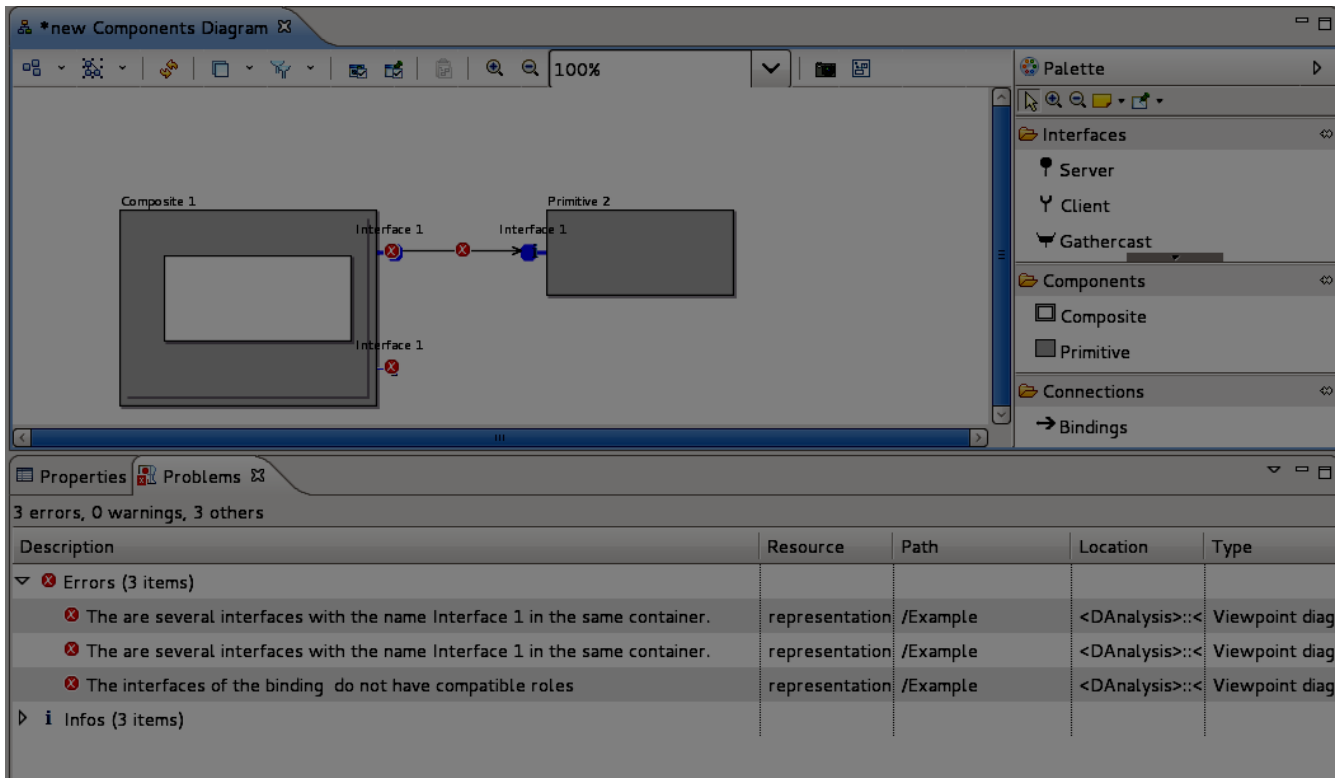
In VCE v.3 it is possible to check some of the validation rules on a VCE Components Diagram. In order to do this open a VCE Components Diagram and select “**Diagram -> Validate**” in the menu.



The elements of the diagram which did not pass the validation should be marked with red

singes. You can also see the validation results in the “**Problems**” tab.

In our example we got the following result:



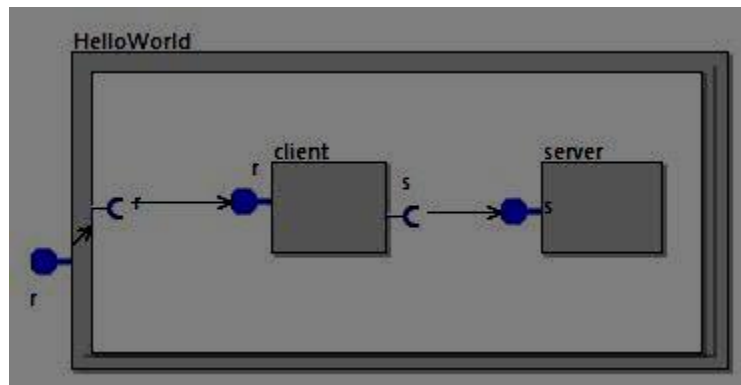
The interfaces of Composite1 did not pass the validation because they have the same names.

The binding did not pass the validation because it goes from a server interface to a server interface.

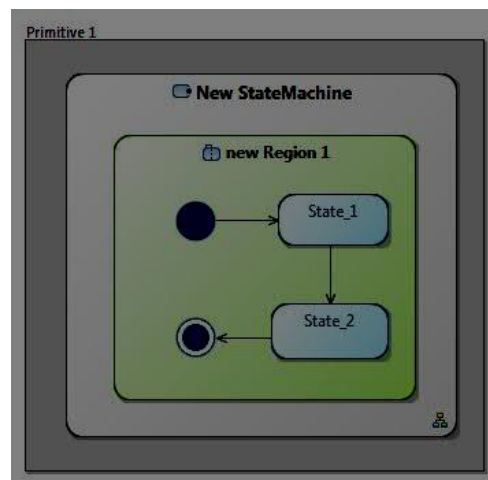
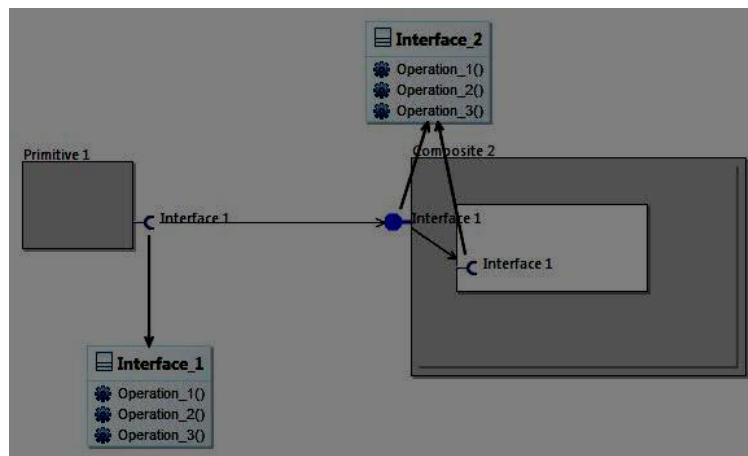
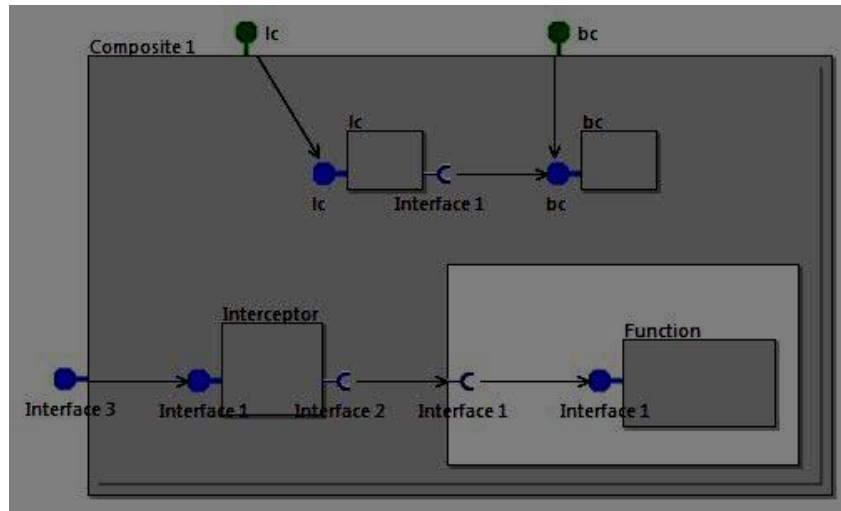
There are not a lot of rules that are checked in the current version of the VCE v.3.

## 14. Examples of diagrams created with VCE v.3

The examples of simple diagrams created with VCE v.3 are given below.







Also, more complicated diagrams can be created using VCE v.3: