Technische Universität München
Fakultät für Informatik

Master Thesis

Modellbasierte Prozessunterstützung für AutoFocus

David Pasch

Aufgabensteller: **Prof. Dr. Manfred Broy**
Betreuer: **Dr. Bernhard Schätz,**
Dipl. Inf. Tobias Hain
Abgabedatum: **15.06.2005**

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.06.2005

____

(David Pasch)

Contents

1 Introduction **1**
1.1 Motivation . . . 3
1.2 Problem Definition . . . 4
1.3 Basic Concepts . . . 5
1.3.1 CCL and ODL . . . 5
1.3.2 Model based Process Support . . . 9
1.4 Document Structure . . . 10

2 Design and Implementation **11**
2.1 Used AutoFocus 2 Subsystems . . . 11
2.1.1 The ODL Interpreter . . . 11
2.1.2 The Window System and Repository Management . . . 13
2.2 Extension of the ODL Editor . . . 14
2.2.1 Original Result Presentation . . . 14
2.2.2 The EvaluationResultWindow . . . 15
2.2.3 Handling of the Product Model . . . 17
2.2.4 Dialog and Window Templates . . . 17
2.2.5 Conclusion . . . 20
2.3 Model based Process Support . . . 20
2.3.1 The Process Model . . . 21
2.3.2 Process Definition . . . 26
2.3.3 Process Application . . . 28
2.3.4 Conclusion . . . 30

3 Application **33**
3.1 Requirements Engineering . . . 33
3.2 Architectural Design . . . 35
3.3 Component Implementation . . . 40

3.4 Code Generation 40

4 User Guide **41**
4.1 Process Definition 41
4.1.1 The Process Edit Dialog 41
4.1.2 The Phase Edit Dialog 43
4.1.3 The Condition and Activity Edit Dialog 43
4.2 Process Application 43
4.2.1 The Phase Menu 47
4.3 File Management 52

5 Conclusion **57**
5.1 Possible Future Extensions 57

A The ODL Grammar **59**

B The Process Model **67**

C The Example Process **69**
C.1 Requirements Engineering (Initial phase) 69
C.1.1 Activities 69
C.2 Architectural Design 71
C.2.1 Conditions 72
C.2.2 Activities 73
C.3 Component Implementation 84
C.3.1 Conditions 85
C.3.2 Activities 86
C.4 Code Generation 97
C.4.1 Conditions 97

Bibliography **99**

Index **101**

Chapter 1

Introduction

Basis of this master thesis is AutoFocus 2 , a modeling tool designed for the specification of embedded and distributed systems. It has been developed at the chairs of Professor Broy and Endres at the TU Munich. Further information can be found on the internet at [AFHOME].

AutoFocus 2 is a graphical editor that allows the user to create and manipulate models interactively. In this document, these models are called product models . A system is modeled as a component that can be decomposed into communicating subcomponents. Each of these subcomponents can be recursively described by such a component mesh again. Consequently, AutoFocus 2 expresses a system as a hierarchy of communicating components. An example is given in Figure 1.1. It shows a screenshot of the application with a product model composed by two connected components labeled “Controller” and “Barometer”.

The task of my bachelor thesis [BachelorThesis] was to extend AutoFocus with a facility to define and evaluate model checks and transformations. For this purpose, [ODL] has introduced the formal languages CCL (Consistency Constraint Language) and ODL (Operation Definition Language). CCL is used to express conditions on the product model and ODL the transformations. In fact, CCL is only a subset of ODL constrained to the features that do not modify the model. An example for a CCL/ODL expression is

```
exists c:Component.
  c.Name = "Controller"
```

which tests for the presence of a component with the name “Controller” in the model. Evaluated on the product model given in Figure 1.1, this expression

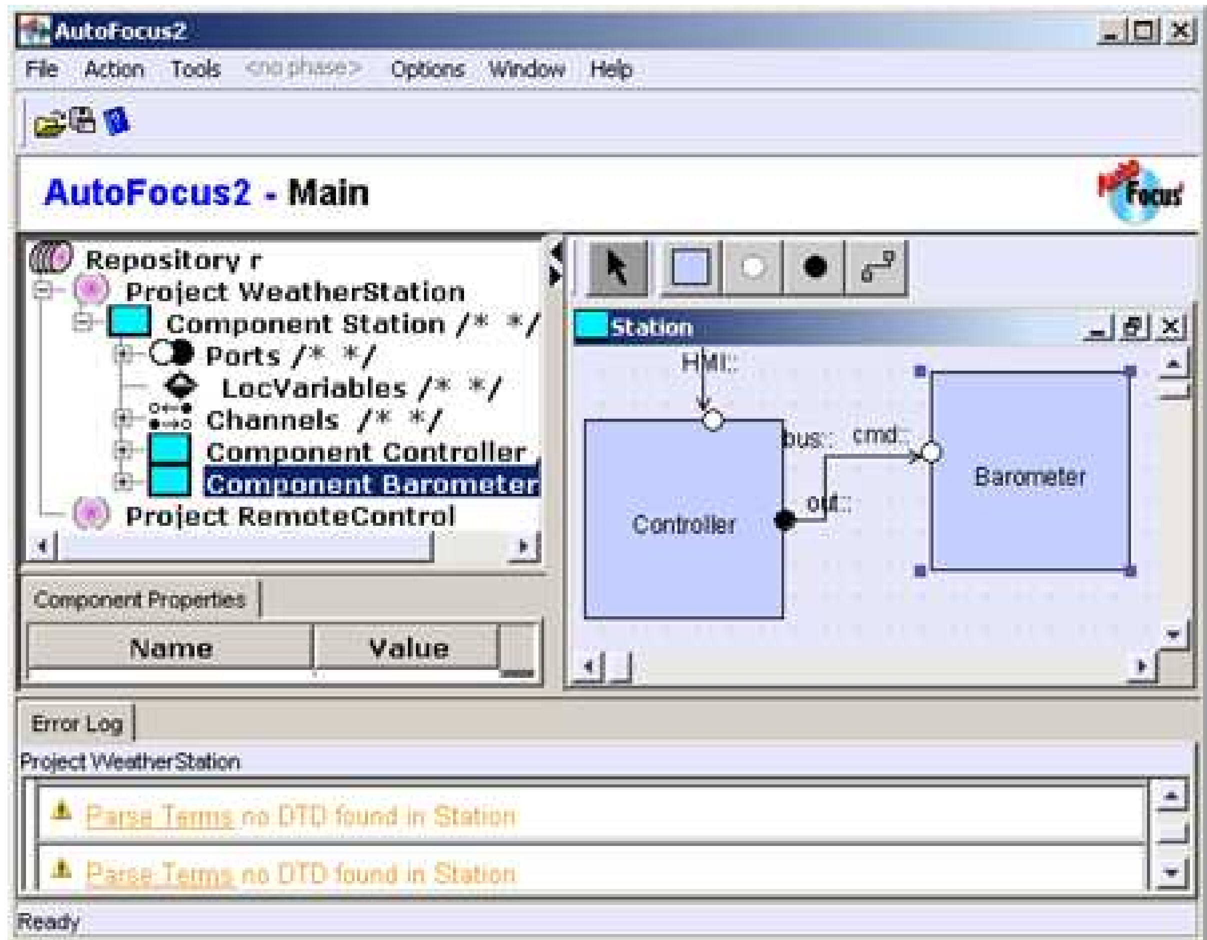


Figure 1.1: AutoFocus 2 application window

yields true as there is in fact a component named "Controller".

The concrete result of the bachelor thesis is an AutoFocus dialog called ODL Editor that allows the user to author, save, and evaluate CCL/ODL expressions on the product model. Optionally, the editor does not only report whether the result of an evaluation is true or false, but it delivers all satisfying or dissatisfying model elements. In the following the satisfying elements are referred to as witnesses and the dissatisfying ones as counter examples . In the above example, the component "Controller" is returned as the only witness.

In this master thesis, AutoFocus 2 shall be extended to allow for model based development as it is described in [ProcessSupport]. [ProcessSupport] introduces the concept of a *process* that structures the work of the user into *phases*. Each phase comes with its own set of model *conditions* and *activities* . The user can only enter a phase if its conditions are satisfied by the product model. Once he has successfully switched to a phase he can access that phase's activities and apply them to the model. The model based development assists the user with his work by ensuring a defined level of quality of the model at all times. We can enforce that the product model is always in a consistent state by expressing the consistency constraints as phase conditions.

Furthermore, by constraining the list of activities to those that are relevant in the respective phase, the overview on the available options is made easier.

For the implementation of the model conditions and activities, this master thesis resorts to the results of the bachelor thesis by defining the conditions and activities as CCL and ODL terms. The Section below motivates the idea behind the model based development processes.

1.1 Motivation

Process models like the V-Model XT are devised to support the development of complex system models. An important feature of these process models is the specification of the products under development like a system specification or system design document, the development phases like analysis or system design, and the qualitative properties which the products shall have in the respective phases. Unfortunately, the processes are usually informally described. Consequently, there are only coarse guidelines for the structure

of the products and what requirements are to be satisfied in what phase. Therefore, little computerized support can be given and the most checks and operations have to be done manually. On the other side, the model based approach allows formal specifications of the products, their quality requirements, and operations on the products.

The use of the AQuA framework enables an automated process support that offers mechanisms for phase oriented checking and transforming of systems. The objective and scope of this thesis are the subject of the next Section.

1.2 Problem Definition

The main goal of this master thesis is the extension of the tool AutoFocus 2 with support for model based development as it is outlined in [ProcessSupport]. It must be possible to define a model based development process and apply it on a product model . An example process shall be devised to demonstrate the functionality.

In addition, the textual output of the evaluation result in the ODL Editor shall be replaced by an interactive one.

The deliverables of this thesis are:

- Java sources implementing the requested functionality along with API documentation,
- an UML diagram modeling the model based development process for the tool MMGen (see below), and
- this document.

The requirements for the new functionality are:

The present textual output of the evaluation result in the ODL Editor is to be replaced. The new presentation of the result allows the user to interactively select a witness or counter example and display it in a view of AutoFocus 2.

AutoFocus 2 shall be extended with a facility for defining, saving, and applying a development process on a model of the user's choice. CCL and ODL are used to define the conditions and activities of a phase. The application of a process starts in the phase that has been chosen by the user

as the initial one. The transition to a phase requires that the entry conditions of that phase are satisfied by the model. In the case of a dissatisfied condition, the user can choose to display a selected counter example in a view of AutoFocus 2. Is the transition successful, the activities of the targeted phase become available to the user. Moreover, he can always recheck the phase conditions on the product model to verify that they haven't been invalidated by model transformations.

Finally, the conditions and activities of a phase can be assigned to user defined categories to facilitate their use.

In addition the following pseudo requirements must be satisfied:

- The implementation language is Java.
- The tool MMGen is used to generate the sources for the model of the process.
- The design of the process model must prepare for alternative phase conditions and activities that do not necessarily make use of CCL or ODL.

The next Section provides you with the basics required to understand the rest of this document.

1.3 Basic Concepts

This Section presents the background material that is a prerequisite for the understanding of the subsequent chapters. It is divided into two parts: A brief introduction into ODL and CCL, and a short survey of the model based development.

1.3.1 CCL and ODL

In the following, the structure of CCL/ODL terms, the relation between CCL and ODL, and the product model will be elucidated. A more complete treatment of CCL and ODL can be found in [ODL].

As it has already been mentioned in the introduction, the Consistency Constraint Language CCL and the Operation Definition Language ODL are formal languages devised for the specification of model checks and transformations. In this thesis, CCL and ODL are used to define conditions and

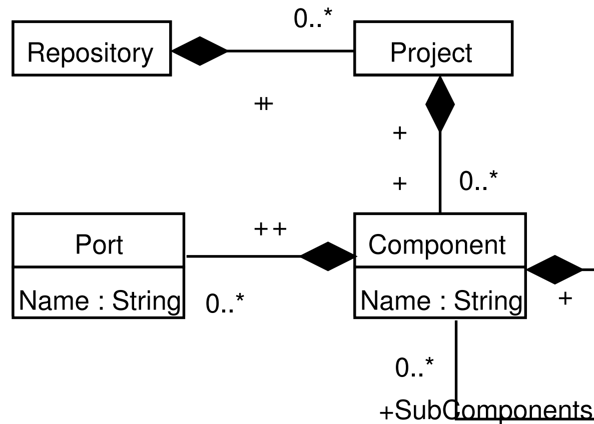


Figure 1.2: View on the structure of a product model

activities on the models created by the AutoFocus user. These models are virtually component hierarchies organized into Projects as it is depicted in Figure 1.2. Thus, to evaluate a CCL or ODL expression, the user has to choose a Project first which identifies the product model to use. For the complete specification of the product model the reader is referred to [AFModel].

On the surface, ODL resembles second order logic. You can denote boolean expressions involving constants and quantified variables combined with the common operators. For example,

```
exists x:Boolean. x
```

and

```
"A" = "B"
```

are valid ODL terms.

A constant or variable can be either of a basic or an extended type. Currently, the basic types are Boolean, Int for integers, String, and the model element types. In the part of the product model structure shown in Figure 1.2, the only model element types are Project, Component, and Port. An extended type is either a set, a product, or a restricted type. An example for a set is

```
set Boolean
```

whose elements are all possible subsets of Boolean — $\{true\}$, $\{false\}$, and $\{true, false\}$. A product combines several types. An example is

```
(c:Component, p:Port)
```

that represents the set of all possible Component, Port pairs. A restricted type builds a set of elements of some type that satisfy a given condition. E.g.,

```
{c:Component | c.Name = "A" or c.Name = "B"}
```

denotes the set $\{``A'' : Component, ``B'' : Component\}$.

Aside the boolean and relational operators we will often need the selector. It provides access to an attribute of the respective object. For example,

```
c.Name
```

yields the name of the Component c. A similar construct is the relation that tests for the presence of an association between two model elements . For example,

```
is Ports(c,p)
```

tests whether the Port p is a port of the Component c.

Variables can be introduced with the following three quantifiers: the existential *exists*, the universal *forall*, and the *context* quantifier. The quantification is always of the form

```
<Quantifier> x : <Type> . <Term>
```

where $< Quantifier >$ is one of the three quantifiers, x and $< Type >$ are the name and type of the bound variable, and $< Term >$ is the quantified ODL term. The context quantifier queries the user for an element of the respective type to determine the value of the bound variable.

With the knowledge we have gained so far, we are able to express conditions on the model. A test for a Component with a Port named "X", for example, can be written as

```
exists c:Component.
  exists p:{p:Port | is Ports(c,p)}.
    p.Name = "X"
```

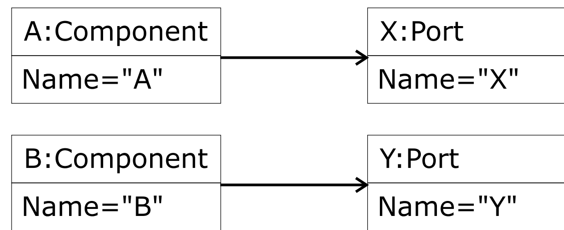


Figure 1.3: An example model

Evaluated on the model shown in Figure 1.3 results in true with the only satisfying assignment or witness $\{c = “A'', p = “X''\}$. If we change the above test to request that all components must have a port X, it will not be satisfied anymore by the model as there is a dissatisfying assignment or counter example $\{c = “B'', p = “Y''\}$.

Up to this point, we have not been able to express modifications to a product model. To create a new element in the product model, you use the so called post universe model type which is denoted by the keyword “new” followed by a model element type. For example,

```
exists c:new Component. true
```

creates a new Component in the product model. To change an attribute or association we have the constructs

```
result has <Relation>
```

and

```
result not has <Relation>
```

For example, the term

```
forall c:{c:Component | c.Name = "a"}. (
  result not has Name (c,"a") and
  result has Name (c,"A")
)
```

renames all Components from “a” to “A”.

In summary, ODL allows you to express logical formulae with existential and universal quantification. Additionally, there is the context quantifier

that queries the user for the value of the bound variable. With this basic feature set, statements on model element, their attributes, and associations between elements can be made. Further constructs are provided to enable modifications of the product model. In concrete, new model elements can be created, their attributes changed, and associations between elements can be added and removed.

ODL offers several more features in addition to what has been presented here. The complete ODL grammar is provided in Appendix A.

CCL is used for defining conditions on product models. It is the subset of ODL without the context quantifier and the constructs for model modifications.

What follows is a discussion of the model based development.

1.3.2 Model based Process Support

The purpose of this Section is the explanation of the model based development from the perspective of this thesis. For further reading you are referred to [ProcessSupport].

The idea underlying the model based development process is to understand the process of developing a system specification as a manipulation of the model of the system. The process is organized into phases where each phase offers the developer a set of activities to transform the model. Additionally, each phase comes with conditions on the model that are supposed to be invariant throughout that phase. Here the conditions and activities are defined with the model oriented languages CCL and ODL. Figure 1.4 shows the basic structure of the model based development process.

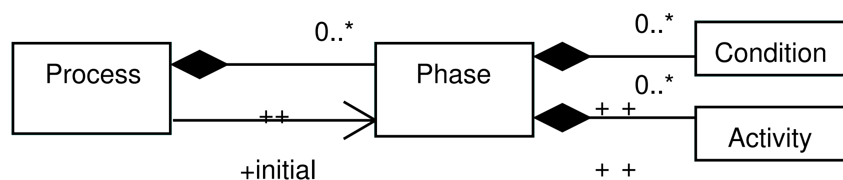


Figure 1.4: Basic structure of a model based development process

The process starts in the initial phase. From there the developer traverses the phases in arbitrary order. In this thesis, the phase conditions are only

checked at the entry of a phase. Hence it is not guaranteed that they are invariant in the respective phase. Instead, the developer is given the option to recheck the conditions himself.

As soon as a phase has been entered, its activities become available to the developer. He can use them to manipulate the product model.

1.4 Document Structure

The next Chapter discusses the changes in the ODL Editor and the added support for model based development processes in AutoFocus 2. It casts light on the implemented subsystems and their relations. In Chapter 3 an example development process is presented and Chapter 4 acts as a user guide for the new process support in AutoFocus 2. It illustrates how development processes are defined and applied with the example process from Chapter 3. Chapter 5 concludes the discussion of the master thesis.

Supplementary material is provided in the Appendices A through C. In Appendix A you will find the grammar of the ODL language and Appendix B shows the UML description of a development process that has been used for the implementation of the process support. Finally, Appendix C provides the definition of the example process that serves in Chapter 4 for the demonstration of the process support.

At the end of this document, the references to external sources are listed in the bibliography and an index for the document is provided.

Chapter 2

Design and Implementation

This Chapter presents the functionality and design of the extensions to AutoFocus 2 that have been implemented in the course of this master thesis. The first Section discusses the parts of AutoFocus 2 that are used by the changed and added subsystems. The second is concerned with the modifications to the ODL Editor and the third with the new subsystem that extends AutoFocus 2 with support for model based development. Throughout this Chapter it is assumed that the reader is familiar with the material presented in Section 1.3.

2.1 Used **AutoFocus** 2 Subsystems

The implementations of the ODL Editor and the model based development resort to other AutoFocus 2 subsystems. This is depicted by Figure 2.1.

In this Section, the used functionality of these subsystems is described. First that of the ODL interpreter followed by those of the window system and repository management.

2.1.1 The ODL Interpreter

As you have seen in Figure 2.1, the ODL interpreter is used by the ODL Editor and the process support for the evaluation of CCL and ODL expressions. The interpreter is interfaced through the Façade Evaluation located in the package quest.odl.evaluation. [AF2API] contains descriptions for the class Evaluation and the others you will see here. For information on the

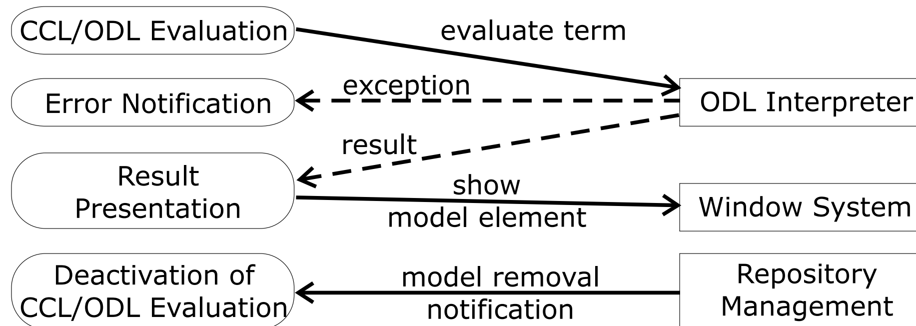


Figure 2.1: Functionality of the ODL Editor and process support (left side) and the used subsystems (right side)

Façade pattern, you are referred to [GammaEtAl]. The relevant parts of the Evaluation class are shown in Figure 2.2.

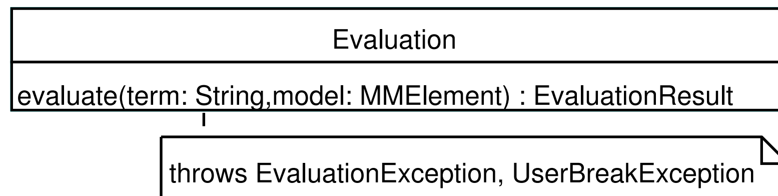


Figure 2.2: Façade to the ODL interpreter

For the evaluation of a CCL or ODL term on a product model the evaluate() method is invoked with the term passed as the first argument and the model as the second. In the case of an ODL term, the specified transformations are applied to the model.

Although, the evaluate() method allows an arbitrary model element (MMElement object) for the representation of the product model, only Projects are currently used for this purpose. The model is given by the aggregation tree rooted at the passed Project.

The result of an evaluation is an EvaluationResult object. It comprises the boolean result value and the set of all variable assignments for the evaluated CCL/ODL term that produce the result value. A single variable assignment is represented by an Iterator that iterates over BoundVariables. A BoundVari-

able represents a bound variable in the evaluated CCL/ODL term with its assigned value. Currently, the value can only be one of the following types: Boolean, Integer, String, Entity , ProductValue , and SetValue .

Entity represents a model element type, ProductValue the product type, and SetValue the set type (cf. Section 1.3.1). Note that the restricted type actually denotes a subset of one of the other types. Therefore, the value always is of one of the previously enumerated types.

Finally, there are a number of errors that can arise during the evaluation. They are signaled by the EvaluationException . It is the base of all exceptions that the interpreter throws except for the UserBreakException . You will receive a UserBreakException when a query initiated by a context quantifier is canceled by the user.

Here we have seen how the ODL interpreter is used — how an CCL or ODL evaluation is triggered, the way the result is structured, and what exceptions can be thrown. Now we will treat the remaining two subsystems that played a major role in the implementation part of this master thesis.

2.1.2 The Window System and Repository Management

This Section deals with the window system and repository management from Figure 2.1. The window system offers the capability to display a model element in the graphical editor of AutoFocus 2 . It is used in the ODL Editor and model based development for showing a witness or counter example to the user. The needed method is found in the class WindowManager in the de.tum.autoflex.base package. It is called openEditor() and takes a model element as argument. On invocation it opens a view in AutoFocus 2 to show the passed element.

The repository management notifies on the removal of the product model . As the model is specified by a Project , the following cases signify such an occurrence. The Project itself is removed, or the repository containing the Project is removed or replaced. In these instances the option to evaluate CCL and ODL terms is disabled in the ODL Editor and process support until a new product model has been chosen.

The RepositoryManager from the de.tum.autoflex.base package notifies its listeners whenever the repository is replaced. To catch the removal of the Repository or Project, you have to register a listener at the Repository.

For more information on the classes and interfaces mentioned herein, you may take a look at the [AF2API].

We have looked at the functionality that is used by the ODL Editor and the process support for showing a model element to the user and for learning the removal of their underlying product model. The rest of this Chapter describes the changes and additions to AutoFocus 2 that have been done in this master thesis.

2.2 Extension of the ODL Editor

Part of this master thesis has been the replacement of the textual result output of the ODL Editor in AutoFocus 2. The new output allows the interactive viewing of the evaluation result. This Section is structured in

- a review of the original textual output,
- its replacement by the EvaluationResultWindow,
- the new handling of the product model,
- the dialog and window templates used by the new dialogs and windows, and
- a summary of this Section.

2.2.1 Original Result Presentation

For a description of the original state of the ODL Editor, you may consult [BachelorThesis]. The main component of the editor is the EvaluationArea . There the user can enter a series of ODL expressions. The choice of the evaluate option of the ODL Editor triggers the evaluation of all of the expressions. Their results are listed in chronological order in the editor's LogWindow . For each evaluation result the boolean result value is output followed by the witnesses or counter examples , respectively.

Below you see an interaction diagram of the objects that are involved in the evaluation and result presentation.

The EditorDialog triggers the evaluation by calling the evaluate() method of its EvaluationArea. The results are returned as Strings in the supplied Vector. It is passed on to the LogWindow for display. The EvaluationArea deals

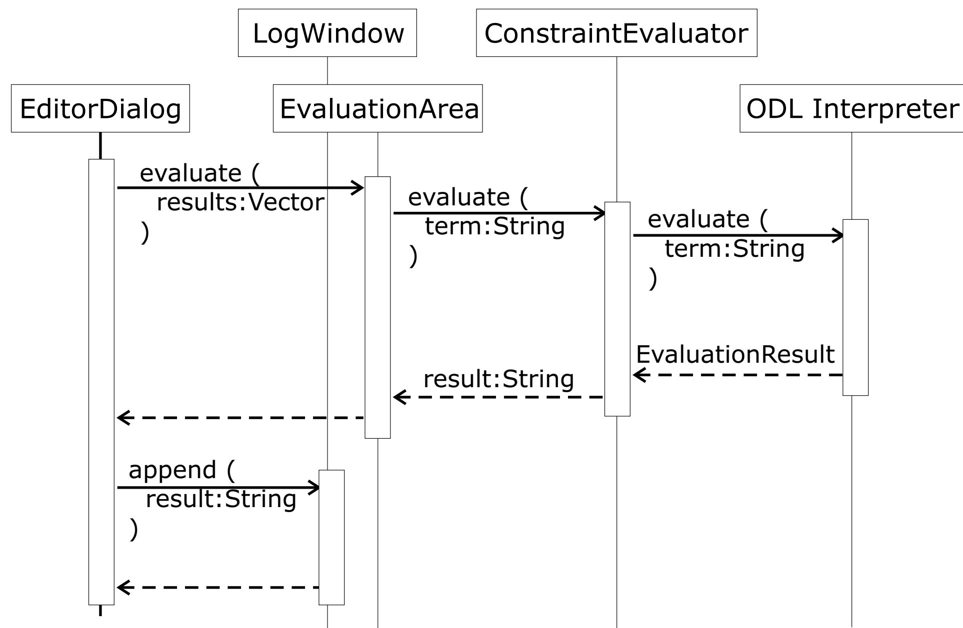


Figure 2.3: Objects responsible for the evaluation and result output

with an evaluate() request by delegating it to the ConstraintEvaluator which evaluates single ODL constraints. The ConstraintEvaluator, in turn, delegates the actual evaluation to the ODL interpreter that returns an EvaluationResult object (cf. Section 2.1.1).

Now we will learn how this system has been changed for the extension of the ODL Editor with an interactive result display.

2.2.2 The **EvaluationResultWindow**

Although, all of the ODL expressions in the evaluation area are still evaluated, only the result of the last one is shown in the EvaluationResultWindow . As before, the boolean result value and the witnesses or counter examples are displayed. The selection of an example in the EvaluationResultWindow opens a view on the respective component in the AutoFocus 2 application window. For the interactive display, the EvaluationResultWindow needs the Evaluation-Result object itself rather than just a textual representation of it. Therefore, the signatures of the evaluate() methods of the affected classes have been ad-

justed. As you can guess from Figure 2.3, the ConstraintEvaluator 's has been changed to return an EvaluationResult object and the EvaluationArea puts the collected EvaluationResult objects into the result Vector.

In Section 2.1.1 we have seen that the witnesses and counter examples are contained by the EvaluationResult object. The single example is represented by an Example object in the EvaluationResultWindow. It has an boundVariablesIterator() method that delivers the BoundVariables of the Example. We know from Section 2.1.1 that the value of a BoundVariable can be a composite, either a ProductValue or a SetValue . That way, the examples shape a forest where the single tree is conceptually structured as shown in Figure 2.4. The

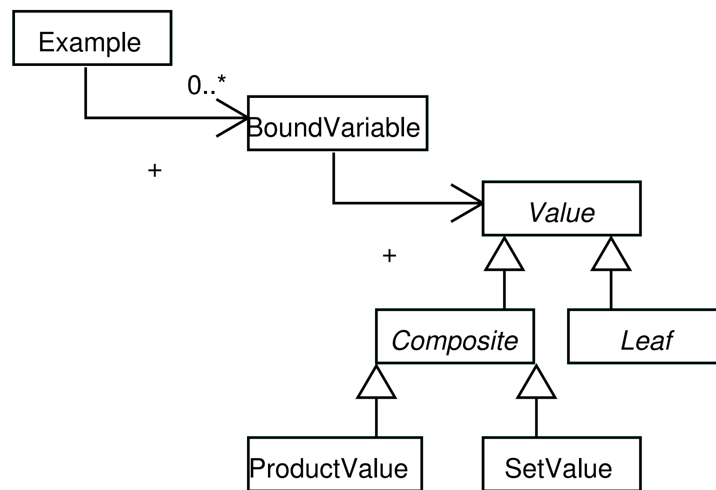


Figure 2.4: Conceptual structure of an Example

only relevant leaves for the EvaluationResultWindow are those of type Entity as a view can be opened on a model element only. Because there was no way of obtaining the model element encapsulated by an Entity object, the accessor getModelElement() has been added to the class Entity (cf. [AF2API]).

Finally, the Show button of the EvaluationResultWindow must open a view in AutoFocus 2 on the model element selected by the user. This is achieved by associating a ShowListener with the Show button which calls the WindowManager 's openEditor() method (cf. Section 2.1.2) on the selected model element when notified. More on the ShowListener in Section 2.2.4.

In this Section we have learned how the EvaluationResultWindow gets its

hands on the EvaluationResult object, how it can unfold it, and open a view on the selected model element.

2.2.3 Handling of the Product Model

Originally the product model was linked to the ODL Editor through selection of the respective Project in the repository browser of AutoFocus 2. To allow the user to interact with the view on a witness or counter example , the editor has been made non-modal. This makes it possible for the user to switch the selected Project or even remove it in the repository browser *while* the ODL Editor is running. For this reason, the way the editor handles the product model has been reworked.

The link to the product model is now established explicitly. Before the editor can be opened, a Project must be chosen in the EditorInvocationDialog . This dialog presents the Projects of the currently loaded Repository to choose from. Its Show button shows the ODL Editor with the product model set to the selected Project.

In addition, the editor must protect itself against the removal of its product model while it is running. For this purpose, a listener is registered at the repository management as it is described in Section 2.1.2. The listener reacts to the removal of the product model by disabling the option to evaluate ODL expressions in the ODL Editor.

All the changes to the functionality of the ODL Editor have been described. The next Section presents the dialog and window templates that have been used for the creation of the EvaluationResultWindow and the EditorInvocationDialog.

2.2.4 Dialog and Window Templates

This master thesis lead to the development of numerous dialogs and windows. To facilitate their management, templates have been introduced that unite their commonalities. These can be understood as applications of the Template pattern that is described in [GammaEtAl].

The dialog and window templates are located in the new package quest.odl.editor.gui.templates. They serve as basis for the new dialogs and windows in the ODL Editor and the process support .

Figure 2.5 puts the new dialog and window of the ODL Editor in relation to the templates.

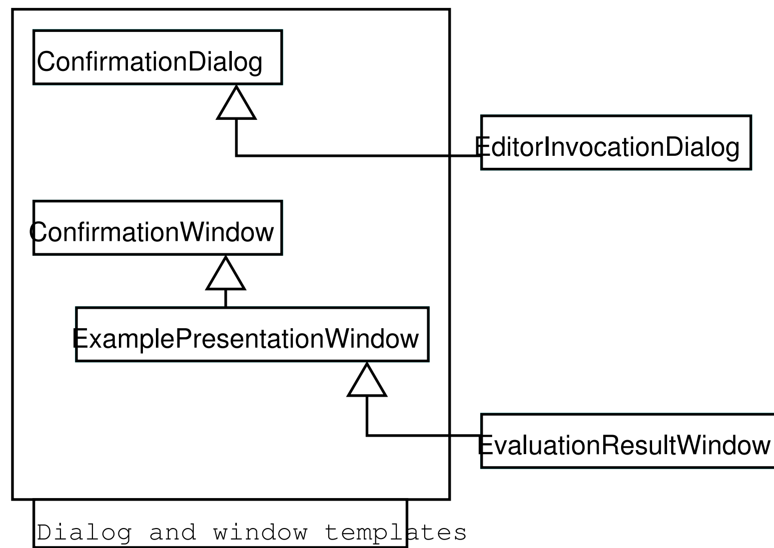


Figure 2.5: The EditorInvocationDialog and EvaluationResultWindow with their templates

The individual templates are discussed in the subsequent subsections.

The ConfirmationDialog and ConfirmationWindow

The ConfirmationDialog/ConfirmationWindow is a dialog/window with an OK and Cancel button for approval and cancellation of an action. Both provide methods for making them (dis)appear, adding components to their content area, and changing the labels of their OK and Cancel button. The procedure for specifying the button actions and learning the user choice differs for the ConfirmationDialog and ConfirmationWindow.

The dialogs based on the ConfirmationDialog are modal — their setVisible() method blocks until the dialog has been closed. The user choice is revealed by the getUserChoice() method as a UserChoice object. The UserChoice is an instance of the enumeration type idiom in Java. The two enumeration values OK and CANCEL are implemented as static fields. The constructor has been made private to prevent the creation of further instances of the class.

The ConfirmationWindow, on the other hand, is non-modal and hence the Observer pattern is employed for the notification of the user choice. A

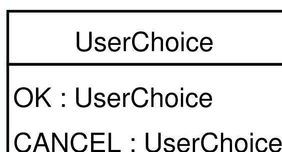


Figure 2.6: The UserChoice class

UserChoiceListener is registered at the window. Its two methods okChosen() and cancelChosen() are invoked in the respective case.

Note that the ConfirmationDialog and ConfirmationWindow will always close on cancel and that in fact the close button in their title bar is interpreted as a cancellation.

The ConfirmationDialog and ConfirmationWindow are very general templates that are the basis for all dialogs and windows introduced in the ODL Editor and process support. They take care of the layout of the components in their content area and the notification of the user choice.

The ExamplePresentationWindow

The ExamplePresentationWindow presents the examples, ie witnesses or counter examples , of an EvaluationResult object. It has a Show button whose action is implemented by a ShowListener . See Figure 2.7 below.

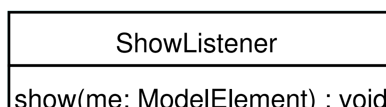


Figure 2.7: The ShowListener interface

On invocation of the show() method, the listener is supposed to present the passed model element to the user.

The EvaluationResultWindow extends this template with the display of the boolean result value (cf. Section 2.2.2).

Hereby all templates have been explained and we conclude the discussion of the changes of the ODL Editor with a final summary of this Section.

2.2.5 Conclusion

One task of the master thesis was the replacement of the textual result output of the ODL Editor by the EvaluationResultWindow. It allows the user to browse through the witnesses or counter examples of the evaluated ODL constraint. He can have it display an individual model element of the examples in a view in the AutoFocus 2 application window. For this purpose, the ODL Editor had to be made non-modal what necessitated a new handling of the underlying product model. While the model has originally been specified through the simple selection of a Project in the repository browser of AutoFocus 2, it is now explicitly linked to the editor in the EditorInvocationDialog. Further, the editor reacts to the removal of its product model by deactivating the evaluation option.

We have seen in this Section how the new EvaluationResultWindow has been integrated into the ODL Editor and how the window system and repository management of AutoFocus 2 have been used for the implementation of the new functionality.

In addition to the EditorInvocationDialog and EvaluationResultWindow, templates for dialogs and windows have been introduced in the new package quest.odl.editor.gui.templates. They serve as basis for the EditorInvocationDialog, the EvaluationResultWindow, and the dialogs and windows for the model based development. While their relations to the EditorInvocationDialog and EvaluationResultWindow have been covered in this Section, those to the dialogs and windows of the process support will be examined in the upcoming Section 2.3.

2.3 Model based Process Support

This Section discusses the new subsystem that extends the tool AutoFocus 2 with support for model based development processes. [ProcessSupport] explains the idea behind the model based development and outlines the functionality that shall be provided by the process support. The user shall be able to define a model based development process and apply it on a product model of his choice. The tasks that resulted include the implementation of the process model (cf. Section 1.3.2), the addition of dialogs for defining, saving, and loading a process and of widgets for applying a process. The following sections describe the products of these tasks.

- *The Process Model* discusses the design and implementation of the model of the development process,
- *Process Definition* presents the new dialogs for defining a process,
- *Process Application* is concerned with the mechanism of applying a process on a product model, and
- *Conclusion* reviews the contents of this Section.

The files belonging to the process support of AutoFocus 2 are found in the new folder af2/Software/ProcessSupport in the cvs repository. The package for the classes is af2.processsupport.

2.3.1 The Process Model

The process model is the foundation of the process support in AutoFocus 2. Figure 2.8 shows the classes of the model which are located in the subpackage model of the processsupport package.

In general we are already familiar with the contents of the Figure from Section 1.3.2. New are the Category and the Check class. The Check class is used by the widgets for the process application and thus will be discussed in Section 2.3.3. Further you may have discovered the clone() methods in the ModelPhase and the Category. They return shallow copies of the objects and are used by the dialogs for the process definition . See Section 2.3.2 for more details.

The following sections elaborate on the Category, conditions, and activities.

The Category

As you have seen in Figure 2.8, the conditions and activities of a phase are organized in category hierarchies. The class design follows the Composite pattern from [GammaEtAl]. The Category class shown in Figure 2.9 takes the role of the Composite whereas its subclasses ModelCondition and ModelActivity are the Leafs — here referred to as atoms.

Consequently, the Category class provides the get, add, and remove methods to deal with its items. The isAtomic() method indicates whether the Category object is an atom or not. In the case of the Category class it returns false, but the ones of the ModelCondition and ModelActivity return true.

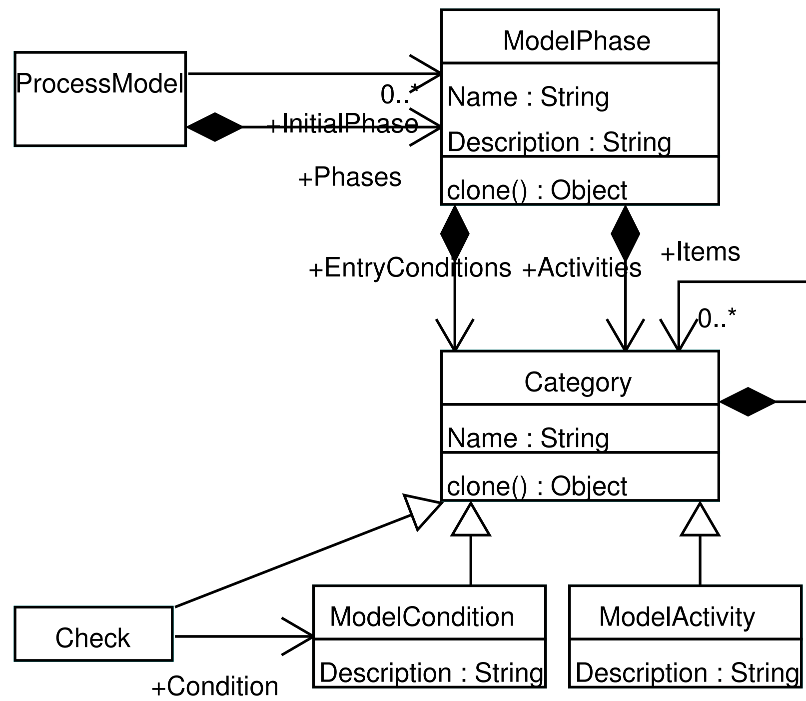


Figure 2.8: View on the process model

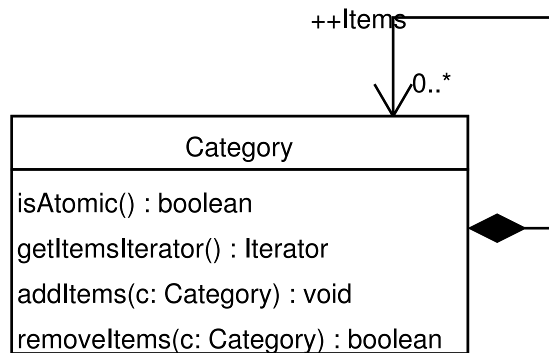


Figure 2.9: The Category class

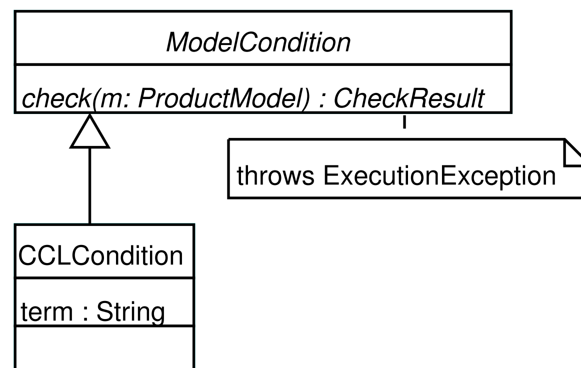


Figure 2.10: Classes that implement the phase conditions

Conditions and Activities

The phase conditions and activities are represented by the abstract classes ModelCondition and ModelActivity , respectively. They are subclassed by CCLCondition and ODLOperation which implement the actual CCL based conditions and ODL based activities. The class diagrams are depicted in Figure 2.10 and 2.11.

The check() and perform() methods evaluate the condition and activity on the supplied product model . In the case of a CCLCondition or ODLOperation the respective CCL or ODL term is evaluated by the ODL interpreter as it has been described in Section 2.1.1. The ProductModel class encapsulates the Project object which serves as the product model (cf. Section 1.3.2). The Project is accessed by the getRoot() method.

The interfaces CheckResult and ActivityResult represent the result of a check or activity, respectively. They are implemented by the classes CCLCheck-Result and ODLOperationResult that are used by the CCLCondition and OD-LOperation. The result of a CCL check comprises the boolean result value and the counter examples retrieved from the EvaluationResult object that has been returned by the ODL interpreter (cf. Section 2.1.1). The result of an ODLOperation tells whether the operation has been aborted by the user. This is signaled by the ODL interpreter with the UserBreakException .

The ExecutionException reports on errors that occurred during evaluation. Its cause (cf. [JavaAPI]) refers to the exception that has been thrown which is either an EvaluationException or an UserBreakException in the case of a

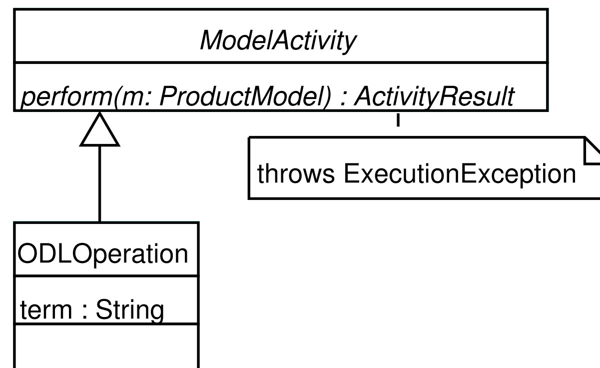


Figure 2.11: Classes that implement the phase activities

CCLCondition.

With this design the process model prepares for future extensions with alternative kinds of conditions and activities as they are announced in [ProcessSupport]. To introduce a new condition or activity, a subclass is added to ModelCondition or ModelActivity and the check() or perform() method is implemented.

By now the reader should have gained an idea of the process model and the role of the ODL interpreter in its implementation. The next Section describes how the code for the process model has been generated.

Code Generation

The diagram below illustrates how the Java sources of the process model have been created.

The basic idea is to generate the Java sources automatically from a given UML diagram. In this case the UML editor ArgoUML has been used to draw the UML diagram of the process model. ArgoUML is available from [ArgoUML]. The diagram is stored in the ProcessModel.zargo file in the ProcessSupport folder and is also provided in Appendix B for reference.

For the code generation the tools DBIExporter and MMGen are used. MMGen is the actual code generator but it requires that the UML diagram is supplied in the dbi format. The DBIExporter is a plugin for ArgoUML that has been used to export the diagram into the ProcessModel.dbi file which is suitable for MMGen. Both tools have been developed at the chair Broy of the TU Munich. Information on the DBIExporter can be found in [DBI]

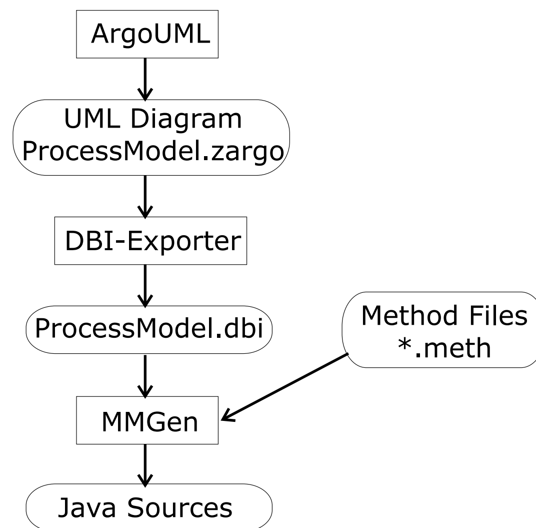


Figure 2.12: Process of source generation

and on MMGen in [MMGen].

From the process model's diagram MMGen creates the Java classes in the package af2.processsupport.model. For the attributes and associations modeled in the UML diagram MMGen automatically generates appropriate set and get methods. Additional methods are provided in so called method files in the ProcessModelMethods folder that are copied by MMGen into the respective classes. There are a number of further features that MMGen automatically adds to the generated classes. Relevant for us are the following:

The implementation of the Visitor pattern for the category hierarchy. That way operations can be defined on the hierarchy without the need to modify the hierarchy classes or to replicate the traversal code. The interfaces and classes that support the Visitor pattern are generated into the package af2.processsupport.visitor. For the Visitor pattern you are referred to [GammaEtAl].

The implementation of the Cloneable interface by the model classes enables the cloning of the ModelPhase , conditions and activities together with their categories.

The implementation of the Serializable interface by the model classes allows the use of the Persistency mechanism of AutoFocus 2 for the saving and loading of the process model to and from a file (cf. [AF2API]).

A description of the Cloneable and Serializable interface is found in [JavaAPI].

With the information on the process model in mind we step into the discussion of the new functionality provided by the process support in AutoFocus 2.

2.3.2 Process Definition

The main task of this master thesis has been the extension of AutoFocus 2 with a facility that allows the definition of a development process and its application. Here we are concerned with the dialogs that have been added for defining a process. They are located in the package af2.processsupport.gui.

In general for every class of the process model there is a form like dialog in which the user edits the attributes and associations of an instance of the class. In concrete, the following dialogs have been added: the ProcessEditDialog , PhaseEditDialog , ConditionEditDialog , and the ActivityEditDialog (cf. Section 2.3.1). Each of these dialogs edits the definition of a part of a process. The name indicates which part it is. The PhaseEditDialog, for instance, edits the definition of a process phase.

Adapters are used for the visual display of the parts of the edited process in the dialogs. Basically, for every class of the process model there is a corresponding adapter in the package af2.processsupport.gui.modelinterface. The ModelPhaseAdapter , for instance, wraps a ModelPhase object. The adapters redefine the toString() method of the adapted object to return a representation suitable for display. Take a look at [AF2API] for the adapters and at [GammaEtAl] for the Adapter pattern.

The dialogs are based on the modal ConfirmationDialog from the quest.odl.editor.gui.templates package (cf. Section 2.2.4). It provides an OK and a Cancel button. On Cancel the object edited in the dialog is to be left unchanged whereas on OK the values entered in the dialog are to be adopted.

All the dialogs work by the same general scheme: Each has a modal show() method that takes the object to be edited as argument. Figure 2.13 depicts the actions take place on invocation of the show() method.

The show() method of the ProcessEditDialog has been chosen here for illustration. The editing occurs on clones of the process' phases. That way, the process object is unchanged in the case of Cancel. On OK, however, the clones replace the original phases and the changes done by the user take effect. OK and Cancel are returned as UserChoice objects by the show() method (cf. Section 2.2.4). The subsequent paragraphs present the peculiarities of each dialog.

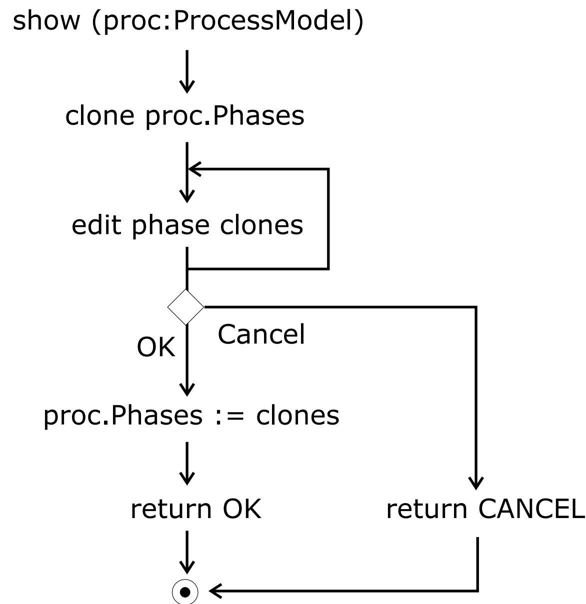


Figure 2.13: Flow chart of the ProcessEditDialog's show() method

The ProcessEditDialog invokes the PhaseEditDialog for defining a new phase or for editing an existent one.

The OK button of the ProcessEditDialog enforces the invariant of the process: An initial phase must be chosen if phases are present (cf. Section 1.3.2).

In analogy to the ProcessEditDialog, the PhaseEditDialog clones the conditions and activities of the supplied phase for the editing. For the addition or editing of a condition (activity) the dialog resorts to the ConditionEditDialog (ActivityEditDialog).

The ConditionEditDialog and ActivityEditDialog share their implementation in the sense of the Template pattern (cf. [GammaEtAl]). They are based on the NamedODLTermEditDialog which does all the work. For this purpose, the adapters for the CCLCondition and ODLOperation share the interface NamedODLTerm that extracts their commonalities. That way, the NamedODLTermEditDialog can be used for both, the CCLCondition and the ODLOperation.

Here we have been concerned with the definition of a process in AutoFocus 2, in the next Section we will look at its application.

2.3.3 Process Application

A process is applied by means of windows based on the ConfirmationWindow from Section 2.2.4 and the phase menu that offers operations related to the current process phase. The widgets interact with the process model through adapters like the dialogs for the process definition do.

This Section is devoted to the description of the widgets. All classes mentioned herein are located in the package af2.processsupport.gui.

Central to the process application facility is the Singleton class ProcessApplicationState that keeps track of the current state of the process application. It is made up of the applied process, the phase the process is currently in, and the product model on which the process is applied. Interested parties are notified on phase transitions in terms of the Observer pattern. The used interface is ProcessApplicationStateListener in af2.processsupport.gui.event. For further information on ProcessApplicationState and the ProcessApplicationStateListener you may consult [AF2API]. The Observer and Singleton pattern are described in [GammaEtAl].

The ProcessApplicationState is set by the ProcessInitiationWindow that allows the user to choose a Project from the AutoFocus 2 repository as the product model. The PhaseTransitionWindow is used to transfer the process into its initial phase. It will be discussed later on.

The ProcessInitiationWindow does not only initialize the ProcessApplicationState, but it also ensures the proper termination of the process by registering listeners at the repository as it is described in Section 2.1.2. When the product model or the process is removed the ProcessApplicationState is reset.

Now let's move on to the PhaseTransitionWindow.

The PhaseTransitionWindow

The PhaseTransitionWindow performs the transition from the current process phase to the supplied one. This is only allowed if all entry conditions of the targeted phase are satisfied by the product model . Then the OK button is enabled which updates the ProcessApplicationState with the new phase.

In the PhaseTransitionWindow the checks of the phase's conditions are modeled as Check objects. Conceptually, a Check object holds the phase condition to be checked with its check result. The condition and the result are represented by a ModelCondition and a CheckResult object, respectively.

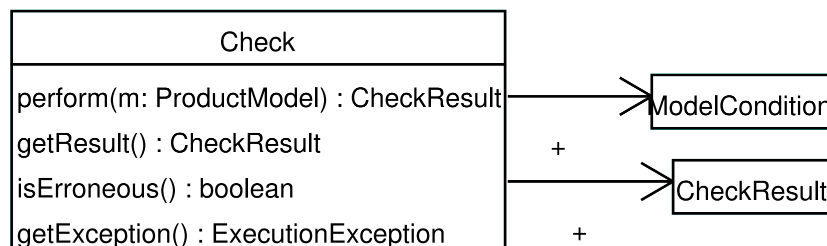


Figure 2.14: The Check class

The perform() method evaluates the condition on the supplied model and sets the result. This is accomplished through delegation to the ModelCondition's check() method. In the case of an ExecutionException the check is marked as erroneous and the exception can be later retrieved with the getExecutionException() method.

The Check class extends the Category class to allow the Check objects to be placed into categories like the phase conditions are (cf. Section 2.3.1). The Visitor support in the process model is used for the creation and evaluation of the Checks. At the invocation of the PhaseTransitionWindow, a Visitor of the phase's entry condition tree constructs a corresponding check tree. When the Check button is pressed, the check tree is traversed by a Visitor that calls the perform() methods of the Check objects. After the evaluation of the Checks, the Inspect button is used to inspect the results of the failed Checks. Is the Check erroneous then the ExecutionException is retrieved and presented in an error message dialog. In the case of a dissatisfied Check, however, the CounterExampleWindow is opened. It is a subclass of the ExamplePresentationWindow that has been described in Section 2.2.4. The CounterExampleWindow uses the functionality of its parent class to present the counter examples of the false CCLCheckResult (cf. Section 2.3.1).

The PhaseTransitionWindow fulfills the important task of changing the phase in the process application. The change is only done if all of the phase's entry conditions are satisfied by the product model. The next Section treats the phase menu which is the central facility for applying a process.

Phase Menu

The phase menu offers the model activities and the checking of the entry conditions of the current process phase. Moreover, it allows the change of the current phase.

On each phase transition the menu is dynamically reconstructed to reflect the new current phase. The notification of the transition is done by a ProcessApplicationStateListener .

For the activities a menu hierarchy is built that matches their category organization. This is done by a Visitor of the category tree that creates the corresponding submenus. The individual phase activity is represented by an atomic menu item which when chosen calls the perform() method of the activity. Should an ExecutionException arise, an error message dialog will be shown with the contents initialized to the message of the exception.

The option for checking the phase conditions in the phase menu leads to the ConditionCheckWindow . It works like the PhaseTransitionWindow with the only difference that the evaluation of the checks is automatically triggered after the opening of the window. Therefore the reader is referred to Section 2.3.3 for a description of the window's operation.

The phase menu's item for switching the current phase opens the PhaseSelectionWindow . There the user chooses the desired target phase from the process' phases. Once the choice has been made, the PhaseTransitionWindow is invoked to actually perform the transition.

With the phase menu we have covered the last piece of the process support in AutoFocus 2. What follows is the conclusion of this Section.

2.3.4 Conclusion

Herein we have talked about the implementation of the support for model based processes in AutoFocus 2. The key classes with their interfaces and relations have been presented. They can be roughly grouped into classes that model a part of a process, dialogs for defining a process, and widgets for applying a process.

The conceptual model of the process from Section 1.3.2 has been augmented with further details. Categories have been introduced to organize the conditions and activities of process phases and the use of abstract classes for the phase conditions and activities allows for alternative implementations in the future. Moreover, it has been illustrated how the given tool MMGen

has been employed to generate the process model sources from an UML description.

Then the graphical components for defining and applying a process have been presented. A process is defined by dialogs in which its constituents are specified. The functions for applying the process are made available through the phase menu. It is reconstructed whenever the process phase changes to reflect the current state of the process application. We have looked at the windows for initiating the process application, performing the phase transitions, and checking the phase conditions.

The next Chapter presents an example process that has been developed with the new process support in AutoFocus 2.

Chapter 3

Application

For the illustration of the model based development an example process has been devised. It is supplied in Appendix C.

From Section 1.3.2 we know that a process comprises phases where each phase has its own set of conditions and activities . The example process structures the development of a system into the four phases Requirements Engineering , Architectural Design , Component Implementation , and Code Generation . The development begins in the Requirements Engineering phase in which the developer elicits and models the requirements for the system. Then, in the Architectural Design phase, the system is decomposed into a hierarchy of communicating components. The behaviors of the components are specified through state automata in the Component Implementation phase. Finally, in the Code Generation phase, the code for the system is generated from the behavioral descriptions of its components. The phases with their conditions and activities are explained in the following sections. To be able to follow the discussion, you will likely have to consult [AFModelUML] on the various kinds of model element and their relations that will be mentioned.

3.1 Requirements Engineering

The Requirements Engineering phase is the initial one of the process. It has no entry conditions . In the phase, the two activities Adapt Business Requirement Priority and Adapt Application Requirement Priority are offered. The user selects a Business or Application Requirement whose priority is set to the highest priority of its subrequirements. The ODL term defining the

Adapt Business Requirement Priority activity is

```
context req:BusinessRequirement.
forall mostImportantReq:{s:AbstractApplicationRequirement |
    call highestPrioritySubReq(s, req.ApplicationRequirements)
  }.
  result has Priority (
      req.Priority,
      mostImportantReq.Priority.Priority
    )
```

The context quantifier asks the user for the Business Requirement whose priority is to be adapted. Then, the forall quantifier iterates over the subrequirements with the highest priority and sets the Business Requirement's priority to theirs with the "result has" construct.

The only interesting part is the determination of the subrequirements with the highest priority. The predicate highestPrioritySubReq() tells whether the passed requirement *s* is among the highest priority subrequirements of the Business Requirement *req*. The definition of the predicate is given below.

```
/**
 * Tells whether s is a highest priority sub-requirement in
 * rootReqs.
 */
define highestPrioritySubReq (
  s:AbstractApplicationRequirement,
  rootReqs:set AbstractApplicationRequirement
) := (
  forall reqs:lfp subReqs
    set areq:AbstractApplicationRequirement with (
      exists root:rootReqs. areq = root
      or
      exists subs:subReqs.
      exists sub:(subs.SubApplicationRequirements).
        areq = sub
    ).
    exists t:reqs.
      s = t
    and
```

```
    forall r:reqs.
      s.Priority.Priority >= r.Priority.Priority
).
```

First, the predicate collects all subrequirements in the set *reqs* with the fix point operator lfp. A subrequirement is either a top level subrequirement from *rootReqs* or it is found somewhere below through the SubApplication-Requirements relation.

After that, the predicate confirms that *s* is indeed a subrequirement of the Business Requirement and that it has the highest priority.

The definition of the Adapt Application Requirement is analogous to the one of Adapt Business Requirement. See Section C.1.1.

3.2 Architectural Design

The phase Architectural Design has the entry conditions No Open Business Requirements, Defined Scenarios, No Open Architectural Constraints, No Open Modal Constraints, and Non-empty interface. The CCL terms for the conditions are trivial and all very alike. The term for the Non-empty interface has been picked as representing example.

```
forall c:Component.
  neg isEmpty(c.Ports)
```

The term shall ensure that there is no component without any I/O ports. The forall quantifier tests all components for the presence of ports.

The activities in the Architectural Design phase are Channel Chain Retyping, Component Encapsulation, and Component Extraction. The activity Channel Chain Retyping queries the user for a port and a type. The type is then assigned to the port and to all channels and ports reachable from the given port. The ODL term can be seen below.

```
context port:Port.
context dataDef:DataDef.
exists type:new MIFType. (
  [...]
  exists connectedPorts:lfp connectedPorts set p:Port with (
      p = port or exists q:connectedPorts.
        call areNeighbours(p,q)
```

```
    ).
  forall connectedPort:connectedPorts. (
    result has DefinedType(connectedPort, type) and
    forall ch:{ch:Channel |
      call hasChannel(connectedPort, ch)}.
      result has DefinedType(ch, type)
  )
)
```

The two context quantifiers obtain the port and type from the user. In the term they are referred to as *port* and *type*. Then, the set *connectedPorts* of all ports reachable from *port* is computed with a fix point operator. A reachable port is *port* itself or a port connected to a reachable one. The predicate areNeighbours() is used to decide whether two ports are connected. Once all affected ports have been determined, their types and those of their channels are set with the "result has DefinedType()" expression. There is no great magic behind the two predicates areNeighbours() and hasChannel(). You may review their definitions in Section C.2.2.

The activities Component Encapsulation and Component Extraction are opposite to each other. Component Encapsulation wraps a given set of components in a new component. Component Extraction, on the other hand, flattens a component by replacing it in the component hierarchy by its subcomponents.

The definition for the activity Component Encapsulation looks like this

```
context comps:set Component.
exists container:new Component.
context name:String. (
  result has Name (container,name) and
  run encapsulate (comps, container)
)
```

The set of components to be encapsulated, here called *comps*, and the name for the new component, called *container*, are specified by the user. The actual encapsulation is done by the encapsulate() procedure in two steps.

```
define encapsulate (
  comps:set Component,
  container:Component
```

```
) as (
  run add (comps, container) and
  run rerouteChannels (comps, container)
).
```

First, the components are added to the new component as subcomponents. This is done by the add() procedure. Then, the channels of the components are rerouted through the ports of the container by the rerouteChannels() procedure.

The definition of the add() procedure is quite straightforward.

```
define add (
  comps:set Component,
  container:Component
) as (
  exists parent:{x:Component | call isParent(x,comps)}. (
    result has SuperComponent(container,parent) and
    result has SubComponents(parent,container) and
    forall c:comps. (
      result not has SubComponents(parent,c) and
      result has SuperComponent(c,container) and
      result has SubComponents(container,c)
    )
  )
).
```

The new container is added to the parent of the components while the components are removed from their parent and appended to the container.

The rerouteChannels() procedure first moves the channels that completely lie in the container from the channel list of the parent to the container's. Then, the channels crossing the container's interface are rerouted by the rerouteOutChannel() and rerouteInChannel() procedure.

```
define rerouteChannels (
  comps:set Component,
  container:Component
) as (
  // Move container's channels from parent to container
  exists parent:{x:Component | call isParent(x,comps)}. (
```

```
    forall intraCh: {
        ch:(parent.Channels) |
          call isContainerPort(ch.SourcePort, comps) and
          call isContainerPort(ch.DestinationPort, comps)
      }. (
      result not has Channels (parent, intraCh) and
      result has Channels (container, intraCh)
    )
    and
    // Route inter-channels through container's interface
    forall outCh: {
        ch:(parent.Channels) |
          call isContainerPort(ch.SourcePort, comps) and
          neg call isContainerPort(ch.DestinationPort, comps)
      }.
      run rerouteOutChannel (outCh, container)
    and
    forall inCh: {
        ch:(parent.Channels) |
          neg call isContainerPort(ch.SourcePort, comps) and
          call isContainerPort(ch.DestinationPort, comps)
      }.
      run rerouteInChannel (inCh, container)
  )
).
```

The rerouteOutChannel() procedure reroutes a channel that leaves the container by adding a new interface port to the container and redirecting the channel through that port. A new channel is created that leads from the subcomponent's port to the new interface port of the container and the old channel's source port is set to the interface port.

```
define rerouteOutChannel (
  outCh:Channel,
  container:Component
) as (
  exists interfacePort:new Port. (
    // container's interface port is "clone" of internal port
    result has Name(interfacePort, outCh.SourcePort.Name) and
```

```
    result has Direction (
        interfacePort,
        outCh.SourcePort.Direction
      ) and
    result has Ports(container, interfacePort) and
    exists intraCh:new Channel. (
      // intra channel is "clone" of external out channel
      result has Name(intraCh, outCh.Name) and
      result has Channels(container, intraCh) and
      // route
      result has SourcePort(intraCh, outCh.SourcePort) and
      result has DestinationPort(intraCh, interfacePort) and
      result has SourcePort(outCh, interfacePort)
    )
  )
).
```

The rerouteInChannel() procedure is symmetric to the rerouteOutChannel() procedure. Its definition and that of the trivial predicates isParent() and isContainerPort() are found in Section C.2.2.

The ODL term of the Component Extraction activity is shown below.

```
define unfold (
  container:Component
) as (
  run extractSubComponents (container) and
  result not has SubComponents (
      container.SuperComponent,
      container
    ) and
  run rerouteChannels (container)
).

context container:Component.
  run unfold (container)
```

The user is queried for the component to flatten which is referred to as *container*. Then, the subcomponents are extracted from the container and the container is removed from the component hierarchy. What's left is the

rerouting of the channels by the rerouteChannels() procedure. In general, the procedure is similar to the one of the Component Encapsulation just that it always does the opposite. The intra-channels are moved from the container's channel list to that of the parent and the channels attached to the container's ports are replaced by new ones that directly connect the subcomponents with the outside world.

3.3 Component Implementation

The entry conditions of the Component Implementation phase are Resolved Types, Resolved Terms, Resolved Ports, Port Pattern Correctness, and Port Channel Type Consistency. The first three check that the model interfaces (MIFs) are consistent and the last two ensure consistent typing of closely related objects. Their implementations are straightforward as you can see in Section C.3.1.

There are two activities called State Encapsulation and State Extraction that are analogous to the activities Component Encapsulation and Component Extraction from Section 3.2. Consequently, you may take the discussion in Section 3.2 as a guide for the understanding of the ODL terms.

3.4 Code Generation

The phase Code Generation has only the condition Defined Behavior that requests that all components possess a specification of their behavior. There is not much more to tell about it. See Section C.4.1.

The example process presented here serves in the next Chapter for the demonstration of the new functionality in AutoFocus 2.

Chapter 4

User Guide

In this Chapter, I will explain how the new support for model based development processes in AutoFocus 2 is used. The Chapter is divided into three parts: First, I will describe the dialogs for defining a development process, then I will show you how to apply it on a selected project. Thereafter, we will take a brief look at the new options in the File menu.

In AutoFocus 2, there always is a development process present that can be edited and applied. In the case you have not loaded one yourself, there will be the default empty process. I will use here the example process from the previous Chapter for illustrating the functionality. In the Section below, we will see how we can change the definition of the process.

4.1 Process Definition

You can edit the definition of the currently loaded process after choosing the Edit Process option in the Action menu of AutoFocus 2. The definition of the process is edited by editing the definitions of its constituents in form like dialogs. Here you may want to review the structure of a process in Section 1.3.2. Below the use of the individual dialogs is explained.

4.1.1 The Process Edit Dialog

The Edit Process option from the Action menu leads you to the process edit dialog . This dialog lists the phases of the currently loaded process. In Figure 4.1 you can see the dialog opened on the example process with its

four phases Requirements Engineering , Architectural Design , Component Implementation , and Code Generation .

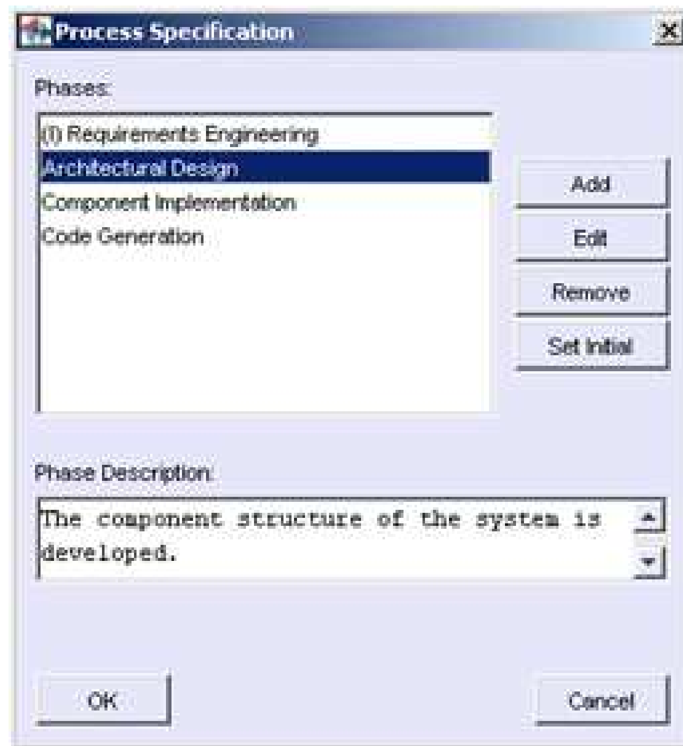


Figure 4.1: The Process Edit Dialog on the example process from Chapter 3

There are buttons for adding new phases to the process, editing the phases' definitions, and removing phases from the process. The Add button adds a new phase to the process. The definition for the phase is entered in the phase edit dialog that will be described later. The Remove button, on the other side, removes the currently selected phase from the process. To edit a phase you can either double click it or press the Edit button. The phase edit dialog is then opened on the phase.

Before the changes to the process definition can be accepted you have to ensure that an initial phase has been chosen with the Set Initial button. The name of that phase will be preceded by an "(I)" in the phase list to mark it as the initial one. You can convince yourself that in the screenshot shown in Figure 4.1 the Requirements Engineering is indeed marked as the initial

phase of the process.

4.1.2 The Phase Edit Dialog

As you can see in Figure 4.2, the phase edit dialog has fields for editing the name and description of the phase. Here it's the Architectural Design phase. The bottom part of the dialog is occupied by the entry conditions and activities of the phase. They are shown in their category hierarchies.

There are buttons for editing the category hierarchies, conditions, and activities. The Add Category, Add Condition, or Add Activity button inserts a new category, condition, or activity into the selected category. The condition or activity is defined in the condition or activity edit dialog . The Remove button is the contrary of the Add buttons. It removes the selected category, condition, or activity from the phase. The Edit button or a double click open the condition or activity edit dialog on the respective condition or activity. These two dialogs are described next.

4.1.3 The Condition and Activity Edit Dialog

The condition or activity edit dialog provides fields for the three attributes of a condition or activity: the name, description, and the CCL/ODL term. See the Figures 4.3 and 4.4.

Here we have seen how a process is defined. In the next Section, we will see how it is applied on a selected product model.

4.2 Process Application

The development process begins with the Start Process item of the Action menu. Thereupon, the process initiation window appears as you can see it in Figure 4.5.

There you choose a project from the repository as the product model for the process. The OK button leads to the phase transition window that conveys the process into the initial phase. We will come back to the phase transition window later. First let's move on to the phase menu .

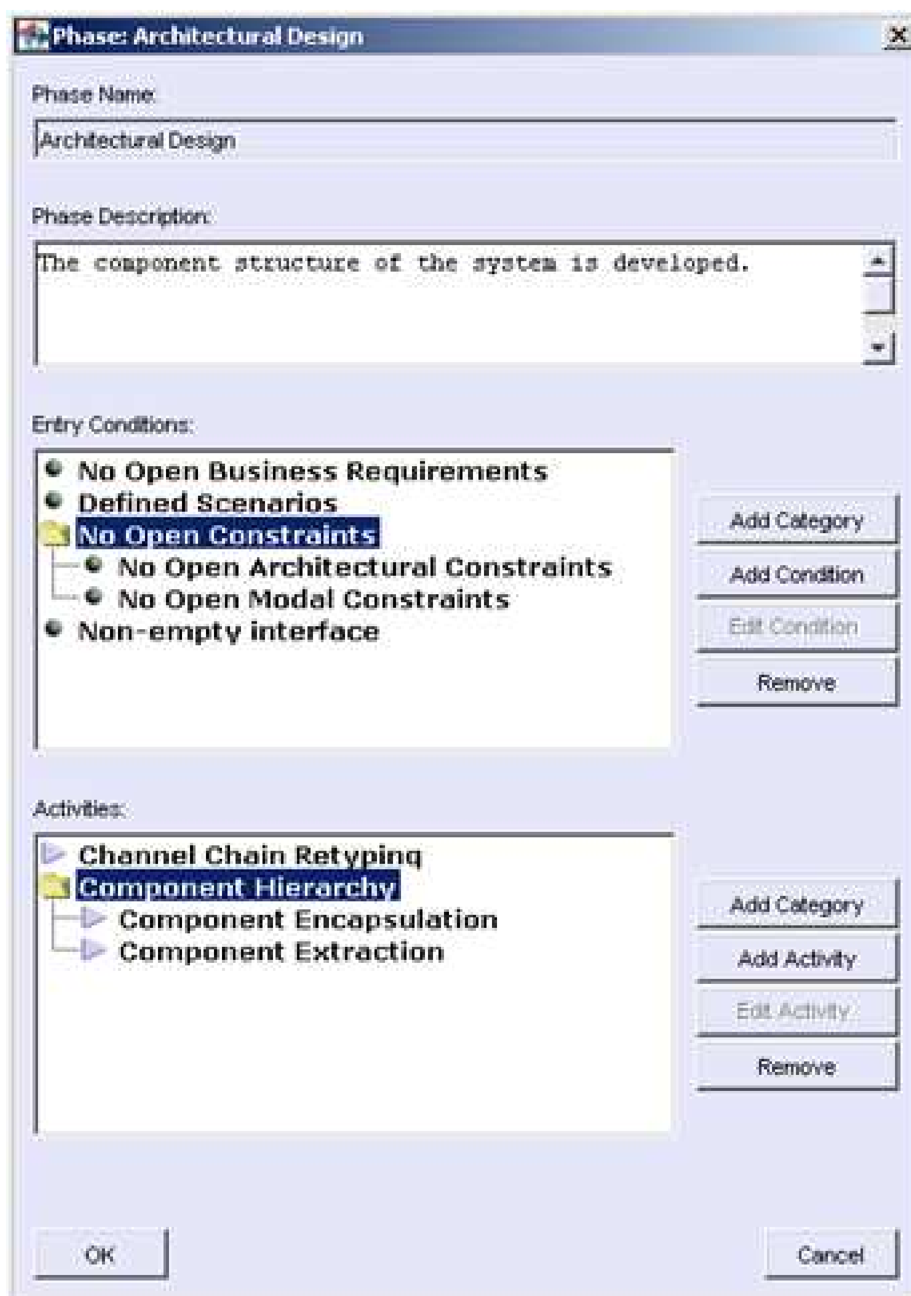


Figure 4.2: The Phase Edit Dialog on the "Architectural Design" phase

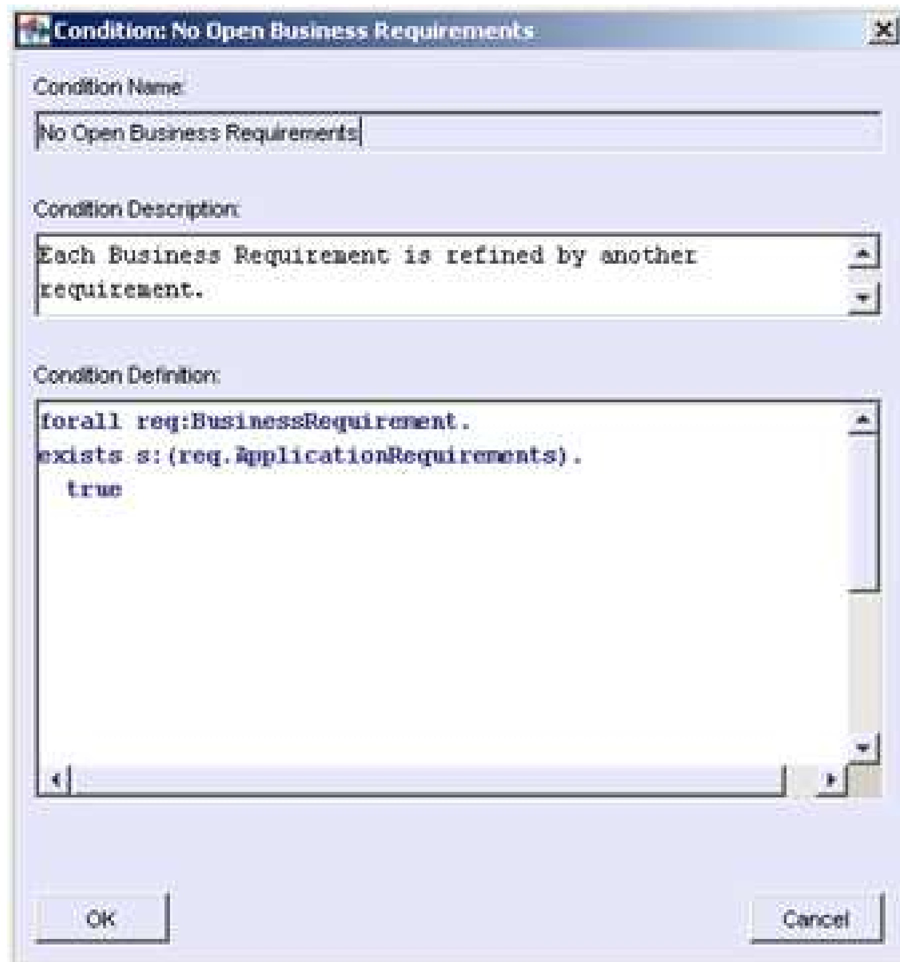


Figure 4.3: The Condition Edit Dialog on the "No Open Business Requirements" condition

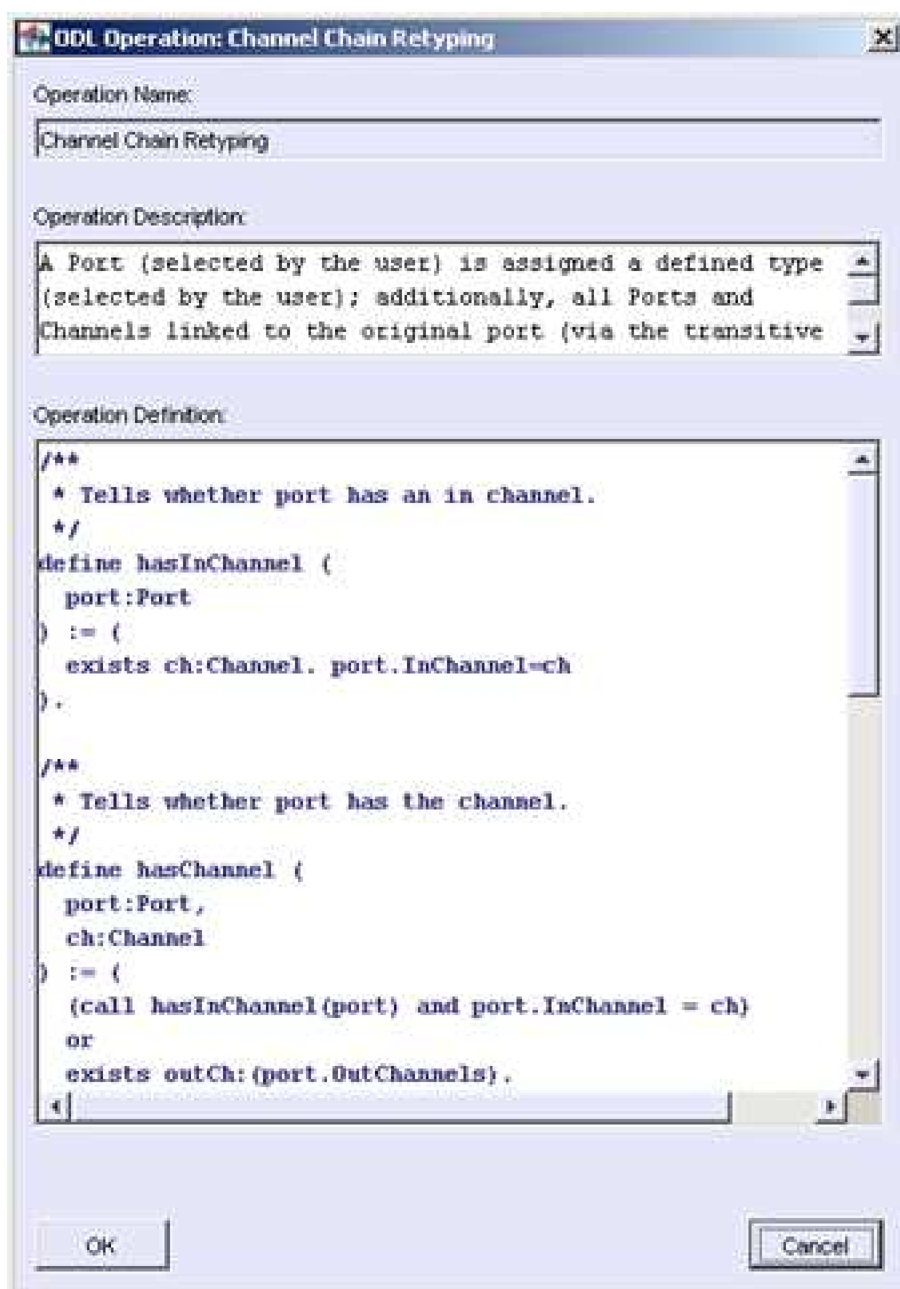


Figure 4.4: The Activity Edit Dialog on the “Channel Chain Retyping” activity

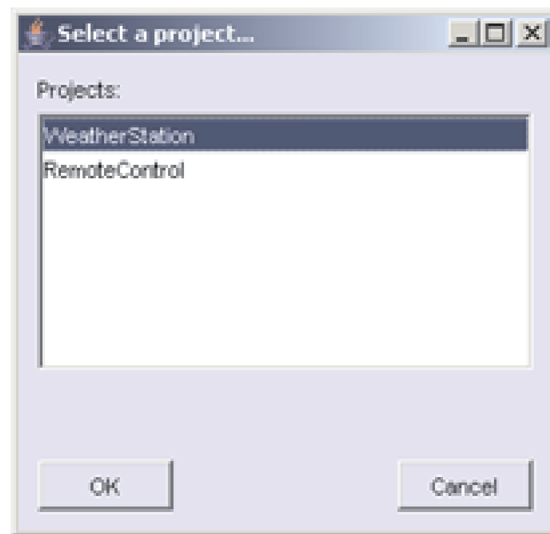


Figure 4.5: The Process Initiation Window

4.2.1 The Phase Menu

The phase menu is the central component for applying the development process. It allows you to advance the process to the next phase and offers operations related to the phase the process is currently in. The menu is shown in Figure 4.6 below.

Figure 4.6: The Phase Menu in the Architectural Design phase

It wears the name of the current process phase. In this case it's "Architectural Design". The functions of the menu items are explained in the following subsections. There are items for switching the phase and for checking the conditions and executing the activities of the current phase.

Switching the Phase

The Switch to Phase item leads to the window shown in Figure 4.7.

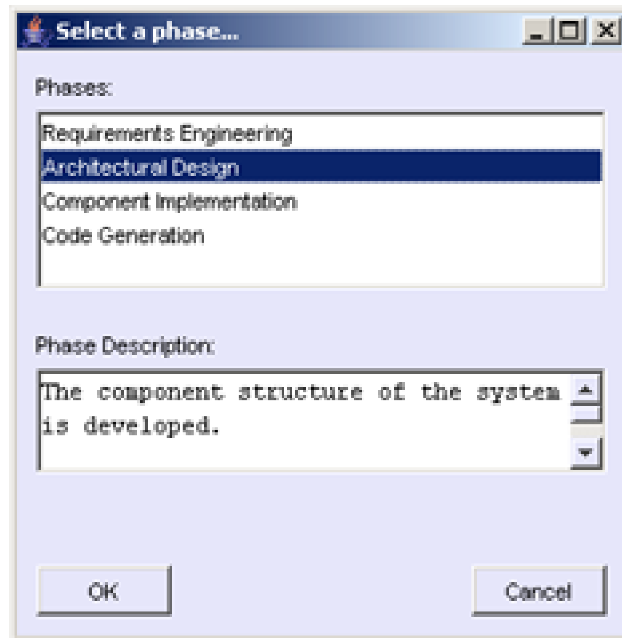


Figure 4.7: Window for choosing the next process phase

There you choose the desired target phase from the process phases. This opens the phase transition window shown in Figure 4.8.

Before the actual transition to the chosen phase can take place, all of the phase's entry conditions must be satisfied by the product model . The checking of the conditions is triggered by the Check button. The lights of the satisfied conditions turn green while those of the dissatisfied turn red. To find out why a condition is not satisfied by the model you can double click it or press the Inspect button. This opens the counter example window that presents the counter examples for that condition. For example, the condition "Non-empty interface" from Figure 4.8 is not satisfied. The corresponding counter example window is shown in Figure 4.9.

The upper half of the window shows the CCL term and the description of the condition. In the lower half, you can see all found dissatisfying variable assignments for the above CCL term. A double click on a model

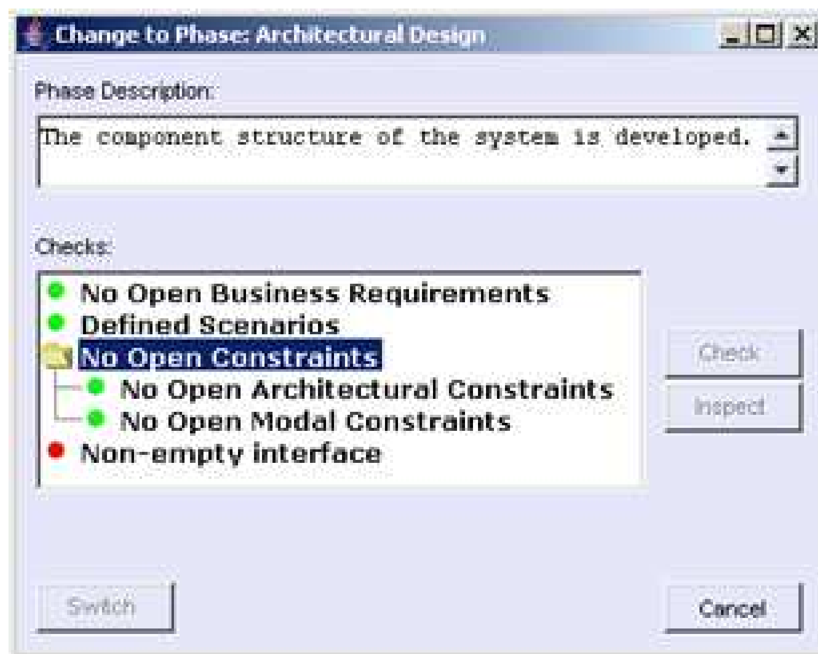


Figure 4.8: The Phase Transition Window with the Architectural Design phase as target

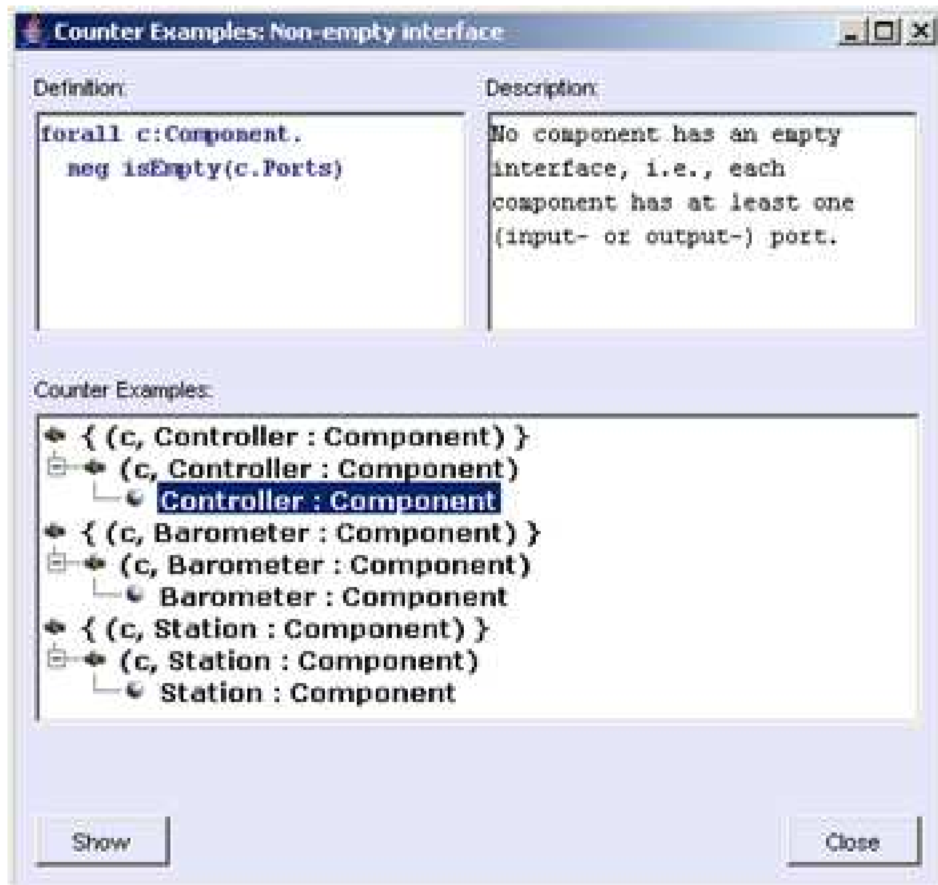


Figure 4.9: The Counter Example Window for the condition "Non-empty interface"

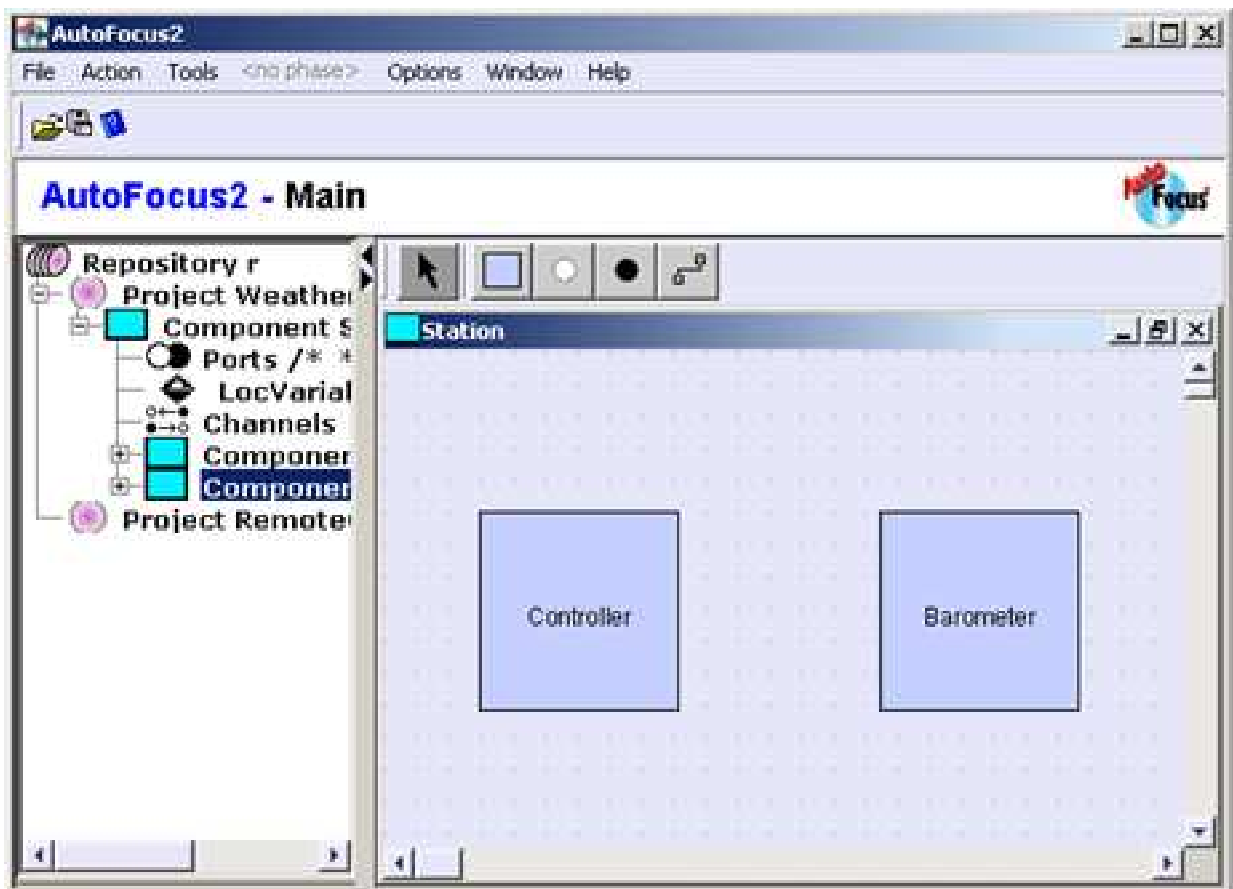


Figure 4.10: View opened for the "Controller" component

element or the press of the Show button displays the element in a view in the AutoFocus 2 application window. For example, the double click of the component "Controller" in Figure 4.9 opened the view in Figure 4.10.

The condition is not satisfied by the component because it has no ports.

Checking the Phase Conditions

To check the conditions of the current phase, you choose the Check Conditions item in the phase menu . This opens the window shown in Figure 4.11.

The lights and the Inspect button work like their equivalents in the phase

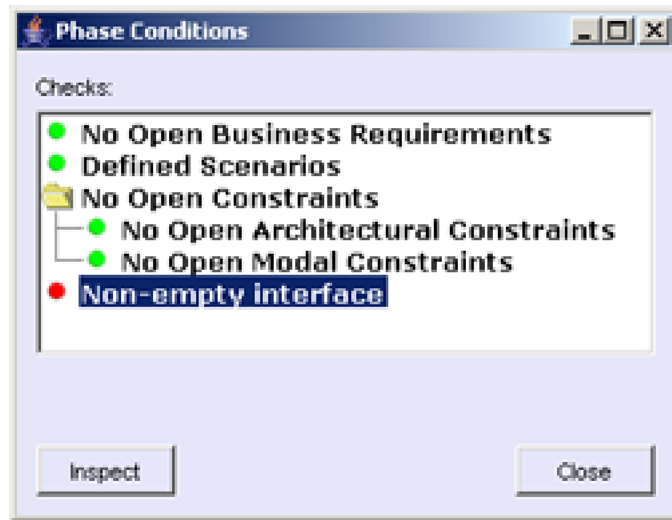


Figure 4.11: Condition Check Window in the Architectural Design phase

transition window . Take a look at the above Section 4.2.1 for details.

Executing Phase Activities

The area in the phase menu below the Check Conditions item is populated with the activities of the current phase. The activities are organized in a menu hierarchy that matches their category hierarchy. The execution of the activity “Component Encapsulation” found in the “Component Structure” category in Figure 4.6, for example, transformed the model from Figure 4.12 into the one in Figure 4.13. The model contains the two components “Controller” and “Barometer” which the activity puts into the new component “X”.

This Section explained how a model based development process is applied in AutoFocus 2. The following Section is about the process related functions in the File menu.

4.3 File Management

The File menu has been extended with options for saving and loading processes, and creating new empty processes. You can reset the current process

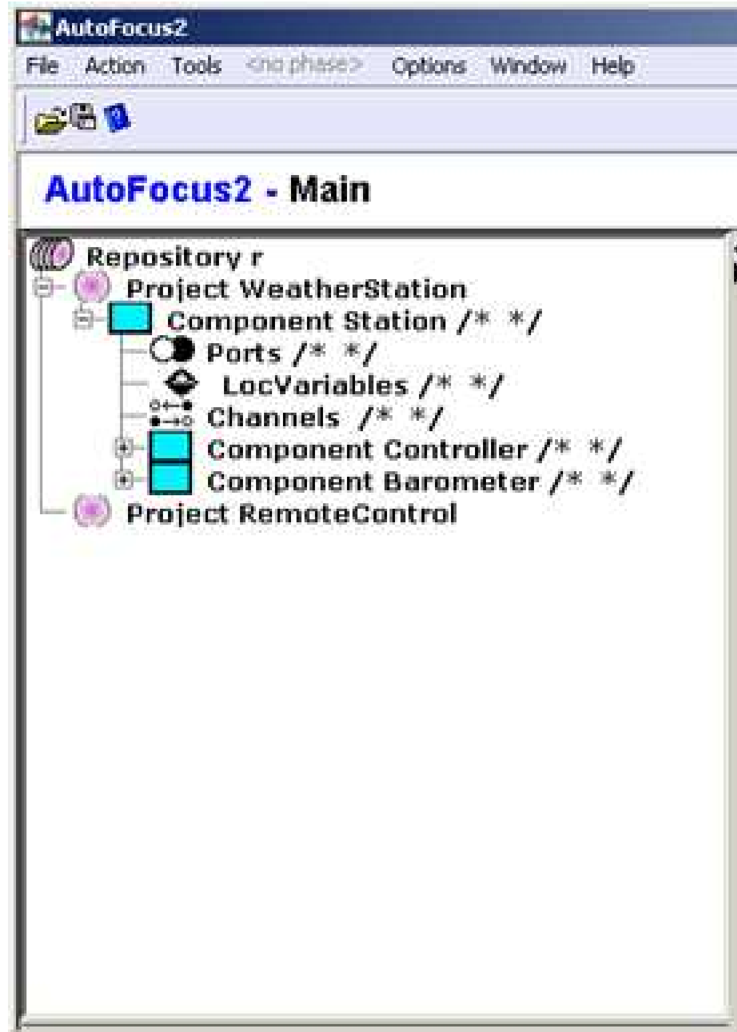


Figure 4.12: Repository browser showing the model before the execution of the "Component Encapsulation" activity

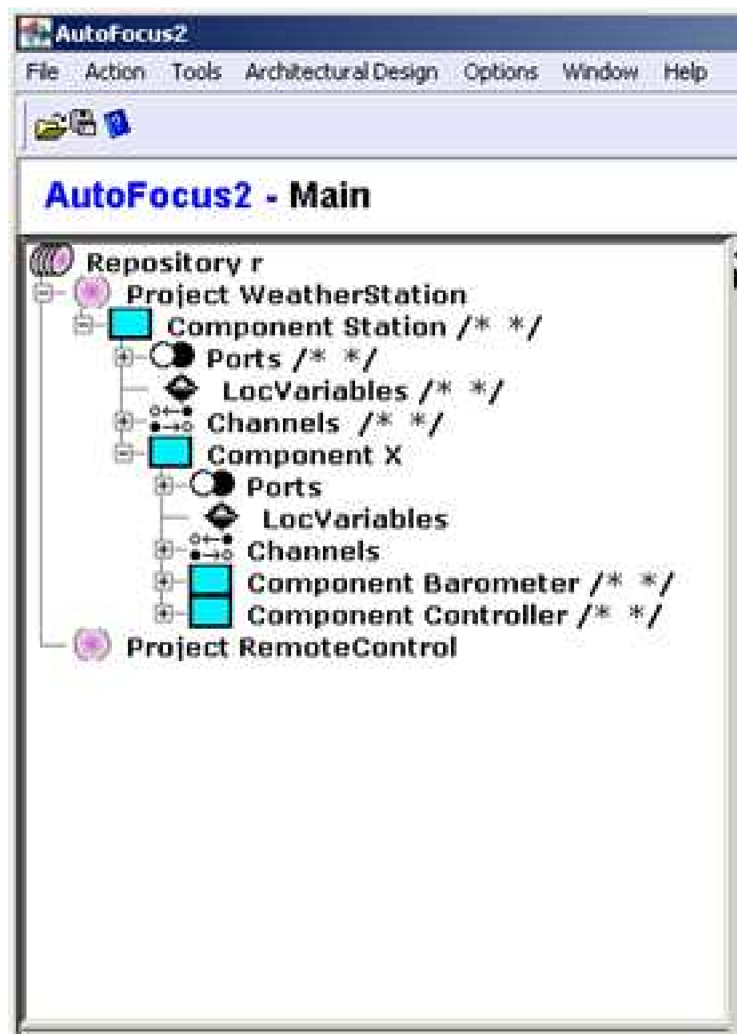


Figure 4.13: Repository browser showing the model from Figure 4.12 after the execution of the "Component Encapsulation" activity

to a new empty one by selecting New→ Process.

With the Open Process item you can load a process from a file you have previously saved with the Save Process or Save Process As item.

The next Chapter concludes this document with a final summary.

Chapter 5

Conclusion

In this document, the results of the master thesis have been presented. The goal of the thesis has been the extension of the model editor AutoFocus 2 with an interactive result output for the ODL Editor and support for model based development processes. The user of the ODL Editor can now have single witnesses or counter examples displayed in a view in the AutoFocus 2 application window. The process support allows the user to define and save model based development processes and apply them on an AutoFocus project. Chapter 2 has described the design and implementation of the changes of the ODL Editor and the process support. For the latter, the development process had to be modeled and implemented by the use of the given code generator MMGen. The definition of the example process that has been developed for the demonstration of the process support has been treated in Chapter 3. The phases together with their conditions and activities have been discussed. For selected conditions and activities the CCL or ODL term has been elucidated. Finally, the use of the process support of AutoFocus 2 has been described in Chapter 4. The individual dialogs for editing the definition of a development process have been shown. The process application has been exemplified with the process from Chapter 4.

5.1 Possible Future Extensions

The here developed process support may be extended in the future with new kinds of conditions and activities which are not necessarily CCL or ODL based. Pointers can be found in [ProcessSupport]. One idea is to enable

conditions and activities to make use of external tools. For instance, a model checker could be invoked by an entry condition of a phase. To introduce a new type of condition or activity, the process model has to be extended as it has been described in Section 2.3.1. A new subclass of ModelCondition or ModelActivity is added and the abstract check() or perform() method is implemented.

Another desired feature may be the enforcement of the model's compliance with the phase conditions during that phase. This would relieve the developer of repeated manual checking of the phase conditions.

At the end of this document, you will find the three appendices providing the ODL grammar, an UML diagram of the process model, and the definition of an example process. In addition, there is a bibliography listing the external references of this document and an index.

Appendix A

The ODL Grammar

Grammar of the Operation Definition Language ODL.

```
odl_start = proposition |
            named_predicate_declaration;

named_predicate_declaration =
  identifier ( bounded_type_list ) := ccl_proposition;

proposition =
  unary_proposition |
  proposition and unary_proposition |
  proposition or unary_proposition |
  proposition implies unary_proposition |
  proposition equiv unary_proposition;

ccl_proposition =
  ccl_unary_proposition |
  ccl_proposition and ccl_unary_proposition |
  ccl_proposition or ccl_unary_proposition |
  ccl_proposition implies ccl_unary_proposition |
  ccl_proposition equiv ccl_unary_proposition;

unary_proposition =
  neg unary_proposition |
  forall_proposition    |
```

```
  exists_proposition    |
  context_proposition   |
  define_statement      |
  ccl_define_statement  |
  run_statement         |
  named_predicate_call  |
  term;

forall_proposition =
  forall bounded_variable_definition . unary_proposition;

exists_proposition =
  exists bounded_variable_definition . unary_proposition |
  exists post_universe_variable_definition . uary_proposition;

context_proposition =
  context context_extension?
    bounded_variable_definition . unary_proposition |
  context context_extension?
    unbounded_variable_definition . unary_proposition;

define_statement =
  define identifier ( bounded_type_list )
    as ( proposition ) . unary_proposition |
  define identifier ( unbounded_type_list )
    as ( proposition ) . unary_proposition ;

ccl_define_statement =
  define identifier ( bounded_type_list )
    assign ( ccl_proposition ) . unary_proposition |
  define identifier ( unbounded_type_list )
    assign ( ccl_proposition ) . unary_proposition ;

ccl_unary_proposition =
  neg ccl_unary_proposition |
  ccl_quantifier bounded_variable_definition .
    ccl_unary_proposition |
  named_predicate_call |
```

```
  ccl_term;

ccl_quantifier = forall | exists;

context_extension = [ hint_extension? ];
hint_extension = hint = string_constant_expr_list;

string_constant_expr_list =
  string_constant_expr string_constant_expr_list_tail*;
string_constant_expr_list_tail = , string_constant_expr;

term = basic_proposition | ( proposition );

ccl_term = ccl_basic_proposition | ( ccl_proposition );

basic_proposition =
  relation |
  bool_proposition |
  functional_proposition ;

ccl_basic_proposition =
  ccl_relation |
  bool_proposition |
  functional_proposition;

functional_proposition = call_expression;

bool_proposition =
  equal_expression |
  bigger_smaller_expression |
  isempty ( expression ) |
  bool_constant_expr;

bigger_smaller_expression =
  expression comparison_operator expression;

comparison_operator = > | < | >= | <=;
```

```
relation =
  pre_relation |
  post_relation;

ccl_relation =
  pre_relation;

pre_relation = is call_expression;

post_relation =
   result has call_expression |
   result not has call_expression;

call_expression = identifier ( args );

named_predicate_call = call call_expression;

run_statement        = run call_expression;

args = arglist?;

arglist = arg arglist_tail*;

arglist_tail = , arg;

arg = expression;

non_constant_expression =
  functional_expression |
  selector_expression |
  defined_variable;

expression =
  non_constant_expression |
  arithmetic_expression |
  constant_expression;

constant_expression =
```

```
  bool_constant_expr |
  string_constant_expr;

bool_constant_expr = bool_constant;

int_constant_expr = sign? int_constant;

string_constant_expr = string_constant;

sign = + | -;

functional_expression = call_expression;

selector_expression = defined_variable . selection;

selection = selector selection_tail*;

selection_tail = . selector;

selector = identifier;

defined_variable = variable;

variable = identifier;

equal_expression = expression = expression;

arithmetic_expression =
  factor |
  arithmetic_expression + factor |
  arithmetic_expression - factor;

factor =
  arithmetic_term |
  factor * arithmetic_term;

arithmetic_term =
  int_constant_expr |
```

```
  ( arithmetic_expression ) |
  size ( expression );

bounded_variable_definition = variable : bounded_type;

unbounded_variable_definition = variable : unbounded_type;

post_universe_variable_definition =
  variable : post_universe_type;

bounded_type =
  bounded_product_type |
  bounded_restricted_type_definition |
  bounded_set_type_definition |
  ( selector_expression ) |
  functional_expression;

bounded_restricted_type_definition =
  { variable : bounded_type | ccl_proposition };

bounded_set_type_definition =
  basic_set_type_definition |
  fixed_point_set_type_definition |
  bounded_map_type_definition;

basic_set_type_definition = set bounded_type;

fixed_point_set_type_definition =
  lfp variable set variable : bounded_type
      with ( ccl_proposition ) |
  gfp variable set variable : bounded_type
      with ( ccl_proposition );

bounded_map_type_definition =
  map variable : bounded_type
    to variable : bounded_type;

bounded_product_type =
```

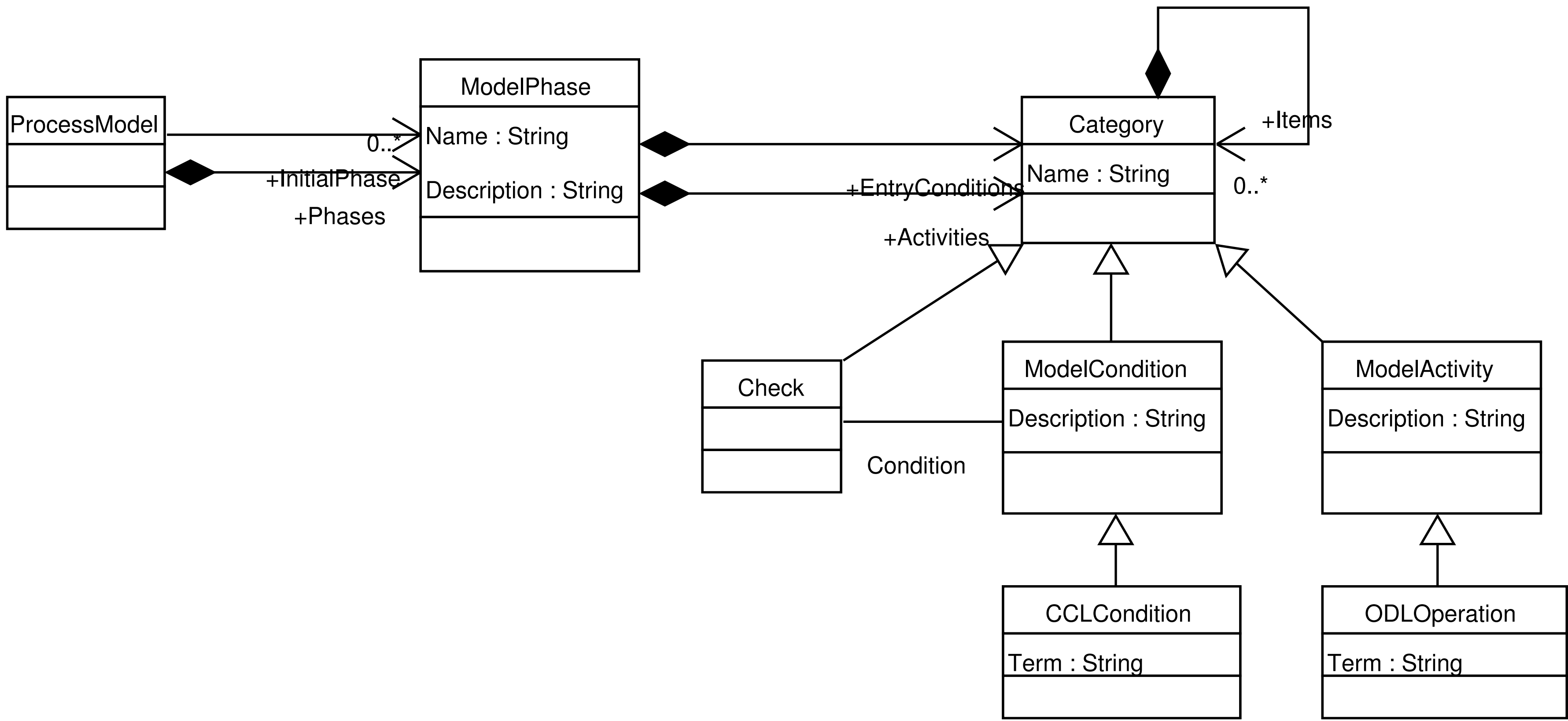
```
  bounded_unary_type |
  ( bounded_type_list );

bounded_type_list = identifier : bounded_type |
  identifier : bounded_type , bounded_type_list;

bounded_unary_type =
  bounded_basic_type |
  model_element_or_defined_variable_type;

bounded_basic_type = bool_type;

model_element_or_defined_variable_type = identifier;

unbounded_type =
  unbounded_product_type |
  unbounded_restricted_type_definition |
  unbounded_set_type_definition;

unbounded_set_type_definition = set unbounded_type;

unbounded_restricted_type_definition =
  { variable : unbounded_type | ccl_proposition };

unbounded_product_type =
  unbounded_unary_type |
  ( unbounded_type_list );

unbounded_type_list =
  identifier : unbounded_type unbounded_type_list_tail* |
  identifier : bounded_type , unbounded_type_list;

unbounded_type_list_tail =
  , identifier : bounded_type |
  , identifier : unbounded_type;

unbounded_unary_type = unbounded_basic_type;
```

```
unbounded_basic_type = int_type | string_type;

post_universe_type =
  post_universe_model_type |
  post_universe_map_type_definition;

post_universe_model_type = new model_element_type;

model_element_type = identifier;

post_universe_map_type_definition =
  map variable : bounded_type
    to variable : post_universe_model_type;
```

Appendix B

The Process Model

The next page shows the process model from the ProcessModel.zargo file found in the ProcessSupport folder in the AutoFocus 2 cvs repository. It has been used for the generation of the process model's source code.


ProcessModel
ModelPhase
Name : String
Description : String
0..*
+InitialPhase
+Phases
Category
Name : String
+Items
0..*
+EntryConditions
+Activities
Check
Condition
ModelCondition
Description : String
ModelActivity
Description : String
CCLCondition
Term : String
ODLOperation
Term : String

Appendix C

The Example Process

Below the phases of the example development process are given. Their conditions and activities are listed together with the CCL and ODL terms that implement them.

C.1 Requirements Engineering (Initial phase)

Description: The process begins with the Requirements Engineering phase. The requirements for the system are modeled.

C.1.1 Activities

Adapt Business Requirement Priority

Description: The priority of a Business Requirement (selected by the user) is adapted to the highest priority of all its subrequirements.

Definition:

```
/**
 * Tells whether s is a highest priority sub-requirement in
 * rootReqs.
 */
define highestPrioritySubReq (
  s:AbstractApplicationRequirement,
  rootReqs:set AbstractApplicationRequirement
```

```
) := (
  forall reqs:lfp subReqs
    set areq:AbstractApplicationRequirement with (
      exists root:rootReqs. areq = root
      or
      exists subs:subReqs.
      exists sub:(subs.SubApplicationRequirements).
        areq = sub
    ). (
    exists t:reqs.
      s = t
    and
    forall r:reqs.
      s.Priority.Priority >= r.Priority.Priority
  )
).

context req:BusinessRequirement.
forall mostImportantReq:{s:AbstractApplicationRequirement |
    call highestPrioritySubReq(s, req.ApplicationRequirements)
  }.
  result has Priority (
      req.Priority,
      mostImportantReq.Priority.Priority
    )
```

Adapt Application Requirement Priority

Description: The priority of an Application Requirement (selected by the user) is adapted to the highest priority of all its subrequirements.

Definition:

```
/**
 * Tells whether s is a highest priority sub-requirement in
 * rootReqs.
 */
define highestPrioritySubReq (
```

```
  s:AbstractApplicationRequirement,
  rootReqs:set AbstractApplicationRequirement
) := (
  forall reqs:lfp subReqs
    set areq:AbstractApplicationRequirement with (
      exists root:rootReqs. areq = root
      or
      exists subs:subReqs.
      exists sub:(subs.SubApplicationRequirements).
        areq = sub
    ). (
    exists t:reqs.
      s = t
    and
    forall r:reqs.
      s.Priority.Priority >= r.Priority.Priority
  )
).

context req:ApplicationRequirement.
forall mostImportantReq:{s:AbstractApplicationRequirement |
    call highestPrioritySubReq (
        s, req.SubApplicationRequirements
      )
  }.
  result has Priority (
      req.Priority,
      mostImportantReq.Priority.Priority
    )
```

C.2 Architectural Design

Description: The component structure of the system is developed.

C.2.1 Conditions

No Open Business Requirements

Description: Each Business Requirement is refined by another requirement.

Definition:

```
forall req:BusinessRequirement.
exists s:(req.ApplicationRequirements).
  true
```

Defined Scenarios

Description: Each Scenario has at least one associated sequence.

Definition:

```
forall sc:Scenario.
  neg isEmpty(sc.Sequence)
```

No Open Architectural Constraints

Description: Each Architectural Constraint is associated with a Component or Channel.

Definition:

```
forall a:ArchitecturalConstraint. (
  neg isEmpty(a.MotivatedChannel) or
  neg isEmpty(a.MotivatedComponent)
)
```

No Open Modal Constraints

Description: Each Modal Constraint is associated with a State or Transition.

Definition:

```
forall mc:ModalConstraint. (
  neg isEmpty(mc.MotivatedState) or
  neg isEmpty(mc.MotivatedTransitionSegment)
)
```

Non-empty interface

Description: No component has an empty interface, i.e., each component has at least one (input- or output-) port.

Definition:

```
forall c:Component.
  neg isEmpty(c.Ports)
```

C.2.2 Activities

Channel Chain Retyping

Description: A Port (selected by the user) is assigned a defined type (selected by the user); additionally, all Ports and Channels linked to the original port (via the transitive closure over Channels and Ports) are assigned the same type.

Definition:

```
/**
 * Tells whether port has an in channel.
 */
define hasInChannel (
  port:Port
) := (
  exists ch:Channel. port.InChannel=ch
).

/**
 * Tells whether port has the channel.
```

```
 */
define hasChannel (
  port:Port,
  ch:Channel
) := (
  (call hasInChannel(port) and port.InChannel = ch)
  or
  exists outCh:(port.OutChannels).
    outCh = ch
).

/**
 * Tells whether the two ports are connected.
 */
define areNeighbours (
  p:Port,
  q:Port
) := (
  (call hasInChannel(p) and p.InChannel.SourcePort = q)
  or
  exists ch:(p.OutChannels).
    ch.DestinationPort = q
).

context port:Port.
context dataDef:DataDef.
exists type:new MIFType. (
  result has Model(type, dataDef.TConst) and
  result has Text(type, dataDef.TConst.Name) and
  exists connectedPorts:lfp connectedPorts set p:Port with (
      p = port or exists q:connectedPorts.
        call areNeighbours(p,q)
    ).
  forall connectedPort:connectedPorts. (
    result has DefinedType(connectedPort, type) and
    forall ch:{ch:Channel |
      call hasChannel(connectedPort, ch)}.
      result has DefinedType(ch, type)
```

```
  )
)
```

Component Encapsulation

Description: A set of Components (selected by the user) is encapsulated by a new Component (with a name defined by the user). The selected Components are introduced as subcomponents of the newly introduced Component. Channels leading to the component are rerouted over newly introduced Ports.

Definition:

```
/**
 * Determines whether parent is a super component of
 * the components in comps.
 */
define isParent (
  parent:Component,
  comps:set Component
) := (
  forall c:comps. c.SuperComponent=parent
).

/**
 * Tells whether port belongs to the container.
 */
define isContainerPort (
  port:Port,
  comps:set Component
) := (
  exists c:comps.
  exists pcon:(c.Ports).
    pcon = port
).

/**
 * Adds comps to container.
```

```
 */
define add (
  comps:set Component,
  container:Component
) as (
  exists parent:{x:Component | call isParent(x,comps)}. (
    result has SuperComponent(container,parent) and
    result has SubComponents(parent,container) and
    forall c:comps. (
      result not has SubComponents(parent,c) and
      result has SuperComponent(c,container) and
      result has SubComponents(container,c)
    )
  )
).

/**
 * Reroutes the out channel from an internal component
 * through a new port of the container.
 */
define rerouteOutChannel (
  outCh:Channel,
  container:Component
) as (
  exists interfacePort:new Port. (
    // container's interface port is "clone" of internal port
    result has Name(interfacePort, outCh.SourcePort.Name) and
    result has Direction (
        interfacePort,
        outCh.SourcePort.Direction
      ) and
    result has Ports(container, interfacePort) and
    exists intraCh:new Channel. (
      // intra channel is "clone" of external out channel
      result has Name(intraCh, outCh.Name) and
      result has Channels(container, intraCh) and
      // route
      result has SourcePort(intraCh, outCh.SourcePort) and
```

```
      result has DestinationPort(intraCh, interfacePort) and
      result has SourcePort(outCh, interfacePort)
    )
  )
).

/**
 * Reroutes the in channel to an internal component
 * through a new port of the container.
 */
define rerouteInChannel (
  inCh:Channel,
  container:Component
) as (
  exists interfacePort:new Port. (
    // container's interface port is "clone" of internal port
    result has Name(interfacePort, inCh.DestinationPort.Name) and
    result has Direction (
        interfacePort,
        inCh.DestinationPort.Direction
      ) and
    result has Ports(container, interfacePort) and
    exists intraCh:new Channel. (
      // intra channel is "clone" of external in channel
      result has Name(intraCh, inCh.Name) and
      result has Channels(container, intraCh) and
      // route
      result has SourcePort(intraCh, interfacePort) and
      result has DestinationPort (
          intraCh,
          inCh.DestinationPort
        ) and
      result has DestinationPort(inCh, interfacePort)
    )
  )
).

/**
```

```
 * Reroutes the channels of the components in comps
 * through the ports of container.
 */
define rerouteChannels (
  comps:set Component,
  container:Component
) as (
  // Move container's channels from parent to container
  exists parent:{x:Component | call isParent(x,comps)}. (
    forall intraCh: {
        ch:(parent.Channels) |
          call isContainerPort(ch.SourcePort, comps) and
          call isContainerPort(ch.DestinationPort, comps)
      }. (
      result not has Channels (parent, intraCh) and
      result has Channels (container, intraCh)
    )
    and
    // Route inter-channels through container's interface
    forall outCh: {
        ch:(parent.Channels) |
          call isContainerPort(ch.SourcePort, comps) and
          neg call isContainerPort(ch.DestinationPort, comps)
      }.
      run rerouteOutChannel (outCh, container)
    and
    forall inCh: {
        ch:(parent.Channels) |
          neg call isContainerPort(ch.SourcePort, comps) and
          call isContainerPort(ch.DestinationPort, comps)
      }.
      run rerouteInChannel (inCh, container)
  )
).

/**
 * Encapsulates comps by container.
 */
```

```
define encapsulate (
  comps:set Component,
  container:Component
) as (
  run add (comps, container) and
  run rerouteChannels (comps, container)
).

context comps:set Component.
exists container:new Component.
context name:String. (
  result has Name (container,name) and
  run encapsulate (comps, container)
)
```

Component Extraction

Description: (Inverse Operation to Component Encapsulation):

A Component (selected by the user) is removed, including its ports; all subcomponents of the component are placed on the same level as the removed Component (i.e. move up one level in the subcomponent hierarchy). Channels leading to those components are rerouted from the deleted ports.

Definition:

```
/**
 * Tells whether port has an in channel.
 */
define hasInChannel (
  port:Port
) := (
  exists ch:Channel. port.InChannel=ch
).

/**
 * Tells whether component owns that port.
 */
define hasPort (
```

```
  comp:Component,
  port:Port
) := (
  exists p:(comp.Ports).
    p = port
).

/**
 * Tells whether port is present in the component set.
 */
define havePort (
  comps:set Component,
  port:Port
) := (
  exists c:comps.
    call hasPort(c, port)
).

/**
 * Extracts sub-components from container.
 */
define extractSubComponents (
  container:Component
) as (
  forall subComp:(container.SubComponents). (
    result not has SubComponents (
        container,
        subComp
  ) and
    result has SuperComponent (
        subComp,
        container.SuperComponent
      ) and
    result has SubComponents (
        container.SuperComponent,
        subComp
      )
  )
```

```
).

/**
 * Bypasses the interface port by creating new in channels
 * from the external source to the internal ports.
 */
define bypassInPort (
  interfaceInPort:Port,
  container:Component
) as (
  forall intInPort: {
      port:Port |
        exists ch:(interfaceInPort.OutChannels).
          ch.DestinationPort = port
        and
          call havePort(container.SubComponents, port)
    }. (
    exists inCh:new Channel. (
      // "clone" in channel
      result has Name(inCh, interfaceInPort.InChannel.Name) and
      // connect outside port with internal one
      result has SourcePort (
          inCh,
          interfaceInPort.InChannel.SourcePort
        ) and
      result has DestinationPort (
          inCh,
          intInPort
        ) and  // kills connection to old internal channel
      result has Channels(container.SuperComponent, inCh)
    )
  )
).

/**
 * Bypasses the interface port by setting the internal
 * port as the source of the outgoing channels.
 */
```

```
define bypassOutPort (
  interfaceOutPort:Port
) as (
  forall outCh:(interfaceOutPort.OutChannels).
    result has SourcePort (
        outCh,
        interfaceOutPort.InChannel.SourcePort
      )
).

/**
 * Removes the ports of the container and reroutes
 * the channels of its sub-components.
 */
define rerouteChannels (
  container:Component
) as (
  // Add intra-channels to parent
  forall intraCh: {
      ch:(container.Channels) |
        call havePort (
            container.SubComponents,
            ch.SourcePort
           ) and
        call havePort (
            container.SubComponents,
            ch.DestinationPort
          )
    }. (
    result not has Channels (
        container,
        intraCh
      ) and
    result has Channels (
        container.SuperComponent,
        intraCh
      )
  )
```

```
and
// Bypass interface ports
forall interfaceInPort: {
    port:(container.Ports) |
      call hasInChannel(port) and
      neg call havePort (
          container.SubComponents,
          port.InChannel.SourcePort
        ) and
      neg call hasPort (
          container,
          port.InChannel.SourcePort
        )
  }. (
    run bypassInPort (
        interfaceInPort,
        container
      ) and
    // eliminate InChannel
    result not has SourcePort (
        interfaceInPort.InChannel,
        interfaceInPort.InChannel.SourcePort
      ) and
    result not has Channels (
        container.SuperComponent,
        interfaceInPort.InChannel
      )
)
and
forall interfaceOutPort: {
    port:(container.Ports) |
      call hasInChannel(port) and
      call havePort (
          container.SubComponents,
          port.InChannel.SourcePort
        )
  }. (
    run bypassOutPort (interfaceOutPort) and
```

```
      // eliminate InChannel
      result not has SourcePort (
          interfaceOutPort.InChannel,
          interfaceOutPort.InChannel.SourcePort
        )
  )
).

/**
 * Unfolds container and replaces it by its mesh.
 */
define unfold (
  container:Component
) as (
  run extractSubComponents (container) and
  result not has SubComponents (
      container.SuperComponent,
      container
    ) and
  run rerouteChannels (container)
).

/**
 * Considers only reasonable cases.
 * It is not expected that input from the inside (outside)
 * to an interface port is send back to the inside (outside).
 */
context container:Component.
  run unfold (container)
```

C.3 Component Implementation

Description: The behaviors of the system's components are defined.

C.3.1 Conditions

Resolved Types

Description: Each MIFType has an associated Type.

Definition:

```
forall mif:MIFType.
exists t:Type.
  mif.Model = t
```

Resolved Terms

Description: Each MIFTerm has an associated Term.

Definition:

```
forall mif:MIFTerm.
exists t:Term.
  mif.Model = t
```

Resolved Ports

Description: Each MIFPort has an associated Port.

Definition:

```
forall mif:MIFPort.
exists p:Port.
  mif.Model = p
```

Port Pattern Correctness

Description: Each Input and Output of a TransitionSegement is well-typed: The Type of the Port and the Type of the Pattern/Expression is identical.

Definition:

```
forall ts:TransitionSegment. (
forall in:(ts.Inputs).
  in.Port.Model.DefinedType.Model =
  in.Pattern.Model.DefinedType
and
forall out:(ts.Outputs).
  out.Port.Model.DefinedType.Model =
  out.Expression.Model.DefinedType
)
```

Port Channel Type Consistency

Description: Each pair of Port and associated Channel has the identical type.

Definition:

```
forall ch:Channel. (
  ch.DefinedType =
  ch.SourcePort.DefinedType and
  ch.DefinedType =
  ch.DestinationPort.DefinedType
)
```

C.3.2 Activities

State Encapsulation

Description: Equivalent Operation as Component Encapsulation, to be applied to states.

Definition:

```
/**
 * Determines whether parent is a super state of
 * the states.
 */
```

```
define isParent (
  parent:State,
  states:set State
) := (
  forall st:states. st.SuperState=parent
).

/**
 * Tells whether interface point belongs to the container.
 */
define isContainerPoint (
  point:InterfacePoint,
  states:set State
) := (
  exists st:states.
  exists ptcon:(st.InterfacePoints).
    ptcon = point
).

/**
 * p := q
 */
define assignPoint (
  p:InterfacePoint,
  q:InterfacePoint
) as (
  result has Name(p, q.Name) and
  result has Direction(p, q.Direction)
).

/**
 * s := t
 */
define assignTransition (
  s:TransitionSegment,
  t:TransitionSegment
) as (
  result has Name(s, t.Name)
```

```
).

/**
 * Adds states to container.
 */
define add (
  states:set State,
  container:State
) as (
  exists parent:{x:State | call isParent(x,states)}. (
    result has SuperState(container,parent) and
    result has SubStates(parent,container) and
    forall st:states. (
      result not has SubStates(parent,st) and
      result has SuperState(st,container) and
      result has SubStates(container,st)
    )
  )
).

/**
 * Reroutes the out transition from an internal state
 * through a new point of the container.
 */
define rerouteOutTransition (
  outTS:TransitionSegment,
  container:State
) as (
  exists interfacePoint:new InterfacePoint. (
    // container's interface point is "clone" of internal point
    run assignPoint (interfacePoint, outTS.SourcePoint) and
    result has InterfacePoints(container, interfacePoint) and
    exists intraTS:new TransitionSegment. (
      // intra transition is "clone" of external out transition
      run assignTransition (intraTS, outTS) and
      result has TransitionSegments(container, intraTS) and
      // route
      result has SourcePoint(intraTS, outTS.SourcePoint) and
```

```
      result has DestinationPoint(intraTS, interfacePoint) and
      result has SourcePoint(outTS, interfacePoint)
    )
  )
).

/**
 * Reroutes the in transition to an internal state
 * through a new point of the container.
 */
define rerouteInTransition (
  inTS:TransitionSegment,
  container:State
) as (
  exists interfacePoint:new InterfacePoint. (
    // container's interface point is "clone" of internal point
    run assignPoint (interfacePoint, inTS.DestinationPoint) and
    result has InterfacePoints(container, interfacePoint) and
    exists intraTS:new TransitionSegment. (
      // intra transition is "clone" of external in transition
      run assignTransition (intraTS, inTS) and
      result has TransitionSegments(container, intraTS) and
      // route
      result has SourcePoint(intraTS, interfacePoint) and
      result has DestinationPoint (
          intraTS,
          inTS.DestinationPoint
        ) and
      result has DestinationPoint(inTS, interfacePoint)
    )
  )
).

/**
 * Reroutes the transitions of the states
 * through the interface points of the container.
 */
define rerouteTransitions (
```

```
  states:set State,
  container:State
) as (
  // Move container's transitions from parent to container
  exists parent:{x:State | call isParent(x,states)}. (
    forall intraTS: {
        ts:(parent.TransitionSegments) |
          call isContainerPoint(ts.SourcePoint, states) and
          call isContainerPoint(ts.DestinationPoint, states)
      }. (
      result not has TransitionSegments (parent, intraTS) and
      result has TransitionSegments (container, intraTS)
    )
    and
    // Route inter-transitions through container's interface
    forall outTS: {
        ts:(parent.TransitionSegments) |
          call isContainerPoint(ts.SourcePoint, states) and
          neg call isContainerPoint (
              ts.DestinationPoint,
              states
            )
      }.
      run rerouteOutTransition (outTS, container)
    and
    forall inTS: {
        ts:(parent.TransitionSegments) |
          neg call isContainerPoint(ts.SourcePoint, states) and
          call isContainerPoint(ts.DestinationPoint, states)
      }.
      run rerouteInTransition (inTS, container)
  )
).

/**
 * Encapsulates states by container.
 */
define encapsulate (
```

```
  states:set State,
  container:State
) as (
  run add (states, container) and
  run rerouteTransitions (states, container)
).

context states:set State.
exists container:new State.
context name:String. (
  result has Name (container,name) and
  run encapsulate (states, container)
)
```

State Extraction

Description: Equivalent Operation as Component Extraction, to be applied to states.

Definition:

```
/**
 * s := t
 */
define assignTransition (
  s:TransitionSegment,
  t:TransitionSegment
) as (
  result has Name(s, t.Name)
).

/**
 * Tells whether state owns that point.
 */
define hasPoint (
  st:State,
  pt:InterfacePoint
) := (
```

```
  exists q:(st.InterfacePoints).
    q = pt
).

/**
 * Tells whether point is present in the set of states.
 */
define havePoint (
  states:set State,
  pt:InterfacePoint
) := (
  exists st:states.
    call hasPoint(st, pt)
).

/**
 * Extracts sub-states from container.
 */
define extractSubStates (
  container:State
) as (
  forall subState:(container.SubStates). (
    result not has SubStates (container, subState) and
    result has SuperState (subState, container.SuperState) and
    result has SubStates (container.SuperState, subState)
  )
).

/**
 * Bypasses the interface point by creating new in transitions
 * from the external sources to the internal points.
 */
define bypassInPoint (
  interfaceInPoint:InterfacePoint,
  container:State
) as (
  forall intInPoint: {
      pt:InterfacePoint |
```

```
        exists ts:(interfaceInPoint.OutSegments).
          ts.DestinationPoint = pt
        and
          call havePoint(container.SubStates, pt)
    }.
  forall inTS:(interfaceInPoint.InSegments).
  exists newInTS:new TransitionSegment. (
    run assignTransition (newInTS, inTS) and
    // connect outside point with internal one
    result has SourcePoint(newInTS, inTS.SourcePoint) and
    result has DestinationPoint(newInTS, intInPoint) and
    result has TransitionSegments (
        container.SuperState,
        newInTS
      )
  )
  and
  // eliminate old transitions between
  // interface point and internal ones
  forall outTS:(interfaceInPoint.OutSegments).
    result not has DestinationPoint (
        outTS,
        outTS.DestinationPoint
      )
).

/**
 * Bypasses the interface point by creating new out transitions
 * from the internal points to the external destinations.
 */
define bypassOutPoint (
  interfaceOutPoint:InterfacePoint,
  container:State
) as (
  forall intOutPoint: {
      pt:InterfacePoint |
        exists ts:(interfaceOutPoint.InSegments).
          ts.SourcePoint = pt
```

```
        and
          call havePoint(container.SubStates, pt)
    }.
  forall outTS:(interfaceOutPoint.OutSegments).
  exists newOutTS:new TransitionSegment. (
    run assignTransition (newOutTS, outTS) and
    // connect outside point with internal one
    result has SourcePoint(newOutTS, intOutPoint) and
    result has DestinationPoint (
        newOutTS,
        outTS.DestinationPoint
      ) and
    result has TransitionSegments (
        container.SuperState,
        newOutTS
      )
  )
  and
  // eliminate old transitions between
  // interface point and internal ones
  forall inTS:(interfaceOutPoint.InSegments).
    result not has SourcePoint(inTS, inTS.SourcePoint)
).

/**
 * Removes the points of the container and reroutes
 * the transitions of its sub-states.
 */
define rerouteTransitions (
  container:State
) as (
  // Add intra-transitions to parent
  forall intraTS: {
      ts:(container.TransitionSegments) |
        call havePoint (
            container.SubStates,
            ts.SourcePoint
          ) and
```

```
      call havePoint (
          container.SubStates,
          ts.DestinationPoint
        )
  }. (
  result not has TransitionSegments (
      container,
      intraTS
    ) and
  result has TransitionSegments (
      container.SuperState,
      intraTS
    )
)
and
// Bypass interface points
forall interfaceInPoint: {
    pt:(container.InterfacePoints) |
      forall inTS:(pt.InSegments). (
        neg call havePoint (
            container.SubStates,
            inTS.SourcePoint
          ) and
        neg call hasPoint (
            container,
            inTS.SourcePoint
          )
      )
  }. (
    run bypassInPoint (interfaceInPoint, container) and
    // eliminate old in transitions
    forall inTS:(interfaceInPoint.InSegments). (
      result not has SourcePoint (
          inTS,
          inTS.SourcePoint
        ) and
      result not has TransitionSegments (
          container.SuperState,
```

```
            inTS
          )
      )
  )
  and
  forall interfaceOutPoint: {
      pt:(container.InterfacePoints) |
        forall outTS:(pt.OutSegments). (
          neg call havePoint (
              container.SubStates,
              outTS.DestinationPoint
            ) and
          neg call hasPoint (
              container,
              outTS.DestinationPoint
            )
        )
    }. (
      run bypassOutPoint (interfaceOutPoint, container) and
      // eliminate old out transitions
      forall outTS:(interfaceOutPoint.OutSegments). (
        result not has DestinationPoint (
            outTS,
            outTS.DestinationPoint
          ) and
        result not has TransitionSegments (
            container.SuperState,
            outTS
          )
      )
  )
).

/**
 * Unfolds container and replaces it by its mesh.
 */
define unfold (
  container:State
```

```
) as (
  run extractSubStates (container) and
  result not has SubStates (
      container.SuperState,
      container
    ) and
  run rerouteTransitions (container)
).

/**
 * Considers only reasonable cases.
 * It is not expected that input from the inside (outside)
 * to an interface point is send back to the inside (outside).
 */
context container:State.
  run unfold (container)
```

C.4 Code Generation

Description: Code is generated for the components.

C.4.1 Conditions

Defined Behavior

Description: Each Component either has a SubComponent or a State or a Mode.

Definition:

```
forall c:Component. (
  neg isEmpty (c.SubComponents)
  or
  exists a:Automaton. (
    c.Automaton = a and
    exists s:State.
      a.State = s
   )
```

```
   or
   exists m:Mode.
     c.SubMode = m
)
```

Bibliography

[AFHOME] Home of AutoFocus:
autofocus.informatik.tu-muenchen.de/

[ODL] Bernhard Schätz. *The ODL Operation Definition Language and the AutoFocus/Quest Application Framework AQuA*. Fakultät für Informatik, TU München.

[AFModel] Peter Braun, Heiko Lötzbeyer and Oscar Slotosch. *Project Quest — Integrated Metamodels for AutoFocus*. Fakultät für Informatik, TU München, October 10, 2001.

[AFModelUML] UML diagram of the AutoFocus product model:
validas.zargo in the folder Metamodel in the AutoFocus 2 cvs repository

[BachelorThesis] David Pasch. *Bachelor Thesis — Konzeption und Implementierung eines ODL-Interpreters für das AutoFocus/Quest CASE-Werkzeug*. Fakultät für Informatik, TU München.

[ProcessSupport] Bernhard Schätz. *Process Support within the AutoFocus/Quest Application Framework AQuA*. Fakultät für Informatik, TU München.

[GammaEtAl] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[AF2API] *AutoFocus 2 API Specification*. Generated by javadoc from the sources.

[JavaAPI] *Java 2 Platform, Standard Edition, v1.3.1 API Specification.*

[ArgoUML] Home of ArgoUML:
argouml.tigris.org/

[MMGen] Peter Braun, Heiko Lötzbeyer and Oscar Slotosch. *Quest — Developers Guide.* Fakultät für Informatik, TU München, March 15, 2002.

[DBI] *DBI-Exporter für ARGO-UML.*

Index

ActivityResult, 23
AutoFocus, 1, 4, 6, 11, 13, 20, 26, 41, 51, 57
BoundVariable, 13, 16
CCLCondition, 23, 27
CCLCheckResult, 23, 29
Category, 21, 29
Check, 21, 28
CheckResult, 23, 28
EvaluationArea, 14, 15
EditorDialog, 14
Entity, 13, 16
Evaluation, 11
EvaluationException, 13, 23
EvaluationResult, 12, 15, 19, 23
Example, 16
ExecutionException, 23, 29, 30
LogWindow, 14
ModelActivity, 21, 23, 58
ModelCondition, 21, 23, 28, 58
ModelPhase, 21, 25, 26
ModelPhaseAdapter, 26
NamedODLTerm, 27
ODLOperation, 23, 27
ODLOperationResult, 23
ODL interpreter, 11, 15, 23
ModelPhase, 21
ProductValue, 13, 16
ProductModel, 23
RepositoryManager, 13
SetValue, 13, 16
UserBreakException, 13, 23
WindowManager, 13, 16
activity— phase, 3, 4, 6, 9, 21, 23, 25, 27, 30, 33, 35, 40, 43, 47, 52, 57
ActivityEditDialog, 26, 43
Architectural Design, 33, 35, 42, 43, 47
bachelor thesis — [BachelorThesis], 1
category, 5, 21, 25, 30, 43, 52
ConditionCheckWindow, 30
ConditionEditDialog, 26, 43
counter example, 3, 4, 8, 12–15, 17, 19, 23, 29, 48, 57
CounterExampleWindow, 29, 48
Code Generation, 33, 40, 42
Component Implementation, 33, 40, 42
condition— phase, 3–5, 9, 21, 23, 25, 27, 28, 30, 33, 35, 40, 43, 47, 51, 57
ConfirmationDialog, 18, 26
ConfirmationWindow, 18, 28
ConstraintEvaluator, 15, 16
EditorInvocationDialog, 17
EvaluationResultWindow, 15, 19
ExamplePresentationWindow, 19, 29

model element, 7, 12, 13, 16, 19, 33, 51
model element type, 6, 13
NamedODLTermEditDialog, 27
ODL Editor, 3, 4, 11, 13, 14, 17, 57
process application, 9, 21, 26, 28, 43, 47, 57
ProcessApplicationState, 28
ProcessApplicationStateListener, 28, 30
process definition, 21, 26, 28, 41, 57
ProcessEditDialog, 26, 41
PhaseEditDialog, 26, 42, 43
phase menu, 28, 30, 43, 47, 51, 52
PhaseSelectionWindow, 30
PhaseTransitionWindow, 28, 30, 43, 48, 52
ProcessInitiationWindow, 28, 43
process model, 3, 9, 20, 21, 24, 26, 28, 29, 58
product model, 1, 3–5, 9, 12, 13, 17, 20, 23, 28, 43, 48
process support, 3–5, 9, 11, 13, 17, 20, 33, 41, 57
Requirements Engineering, 33, 42
ShowListener, 16, 19
Persistency, 25
UserChoice, 18, 26
UserChoiceListener, 19
witness, 3, 4, 8, 12–15, 17, 19, 57

Adapt Application Requirement Priority, 33
Adapt Business Requirement Priority, 33
adapter, 26–28

ArgoUML, 24

basic type, 6

CCL, 1, 5, 9, 11, 14, 23, 35, 43, 48, 57
Channel Chain Retyping, 35
code generation, 24
Component Encapsulation, 35, 52
Component Extraction, 35, 40
Component Implementation, 40
Component Structure, 52
constant, 6
context quantifier, 7, 13, 34, 36

DBIExporter, 24
Defined Behavior, 40
Defined Scenarios, 35

enumeration type, 18
error — CCL or ODL evaluation, 11, 13, 23, 29
evaluation — CCL or ODL, 6, 11, 13, 14, 17, 23
evaluation result — CCL or ODL, 4, 8, 11, 12, 14, 23
example process, 4, 33, 41, 57
exists quantifier, 7
extended type, 6

forall quantifier, 7, 34, 35

initial phase, 5, 9, 27, 28, 33, 42, 43

MIF, 40
MMGen, 4, 24, 57
model based development, 3–5, 9, 11, 13, 17, 20, 33, 41, 57

new keyword, 8

No Open Architectural Constraints, 35
No Open Business Requirements, 35
No Open Modal Constraints, 35
Non-empty interface, 35, 48

ODL, 1, 5, 9, 11, 14, 23, 33, 35, 39, 40, 43, 57

phase, 3, 9, 26, 28, 30, 33, 35, 40, 41, 43, 48, 51, 52, 57
phase transition, 5, 28, 30, 47, 48
Port Channel Type Consistency, 40
Port Pattern Correctness, 40
post universe model type, 8
process — development, 3, 4, 9, 20, 26, 28, 33, 41, 43, 52, 57
product type, 6, 13
project — AutoFocus, 6, 12, 13, 17, 23, 28, 41, 43, 57

relation, 7
repository — AutoFocus, 13, 17, 28, 43
repository management, 11, 13, 17, 28
Resolved Ports, 40
Resolved Terms, 40
Resolved Types, 40
restricted type, 6, 13
result (not) has construct, 8, 34, 36

selector, 7
set type, 6, 13
State Extraction, 40
State Implementation, 40

V-Model XT, 3

variable, 6
view on a model element, 11, 16, 51, 57

window system, 11, 13