DIPLOMA THESIS

Improved Tool Support for Model-Driven Development of Interactive Applications in UCP

Submitted at the Faculty of Electrical Engineering and Information Technology,
Vienna University of Technology,
in partial fulfillment of the requirements for the degree of
Diplom-Ingenieur (equals Master of Sciences)

under supervision of

Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl
Univ.Ass. Dipl.-Ing. Dr.techn. Roman Popp
Univ.Ass. Dipl.-Ing. David Raneburger, BSc

by

Vedran Šajatović
Matr.Nr. 0625447
Waaggasse 2–2A/5/42, 1040 Wien

February 23, 2014 _____

Kurzfassung

Modellgetriebene Software-Entwicklung benötigt adäquate Tool-Unterstützung, um effizient eingesetzt werden zu können. Die bereits bestehende Tool-Unterstützung in der *Unified Communication Platform* (UCP) deckt bereits viele Teile vom GUI-Entwicklungsprozess ab. Einige wichtige Schritte mussten allerdings noch händisch gemacht werden. Das Ziel dieser Arbeit war es, die Tool-Unterstützung in UCP zu verbessern, um den ganzen Prozess abzudecken. Um dieses Ziel zu erreichen wurden im Rahmen dieser Arbeit einige neue Tools zu UCP hinzugefügt und einige der bestehenden Tools verbessert. Durch diese Verbesserungen ist der Umgang mit den UCP-Tools intuitiver geworden und kann neuen GUI-Entwicklern leichter vermittelt werden.

Abstract

Model-driven software development requires adequate tool support in order to be practicable. The existing tool support in the Unified Communication Platform (UCP) already covered many parts of the GUI development process, but some important aspects still had to be done manually. The objective of this work was to improve the tool support in UCP to make it cover the whole process. In order to satisfy this goal, several new tools have been added to UCP and some existing UCP tools have been improved in the course of this work. Based on these improvements, using the UCP tools is now more intuitive and easier to learn for GUI developers.

Table of Contents

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
BPMN	Business Process Model and Notation
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JET	Java Emitter Templates
MB	Megabyte
OCL	Object Constraint Language
RCP	Rich Client Platform
REST	Representational State Transfer
RST	Rhetorical Structure Theory
UCP	Unified Communication Platform
UI	User Interface
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
WSDL	Web Services Description Language
XML	Extensible Markup Language

1 Introduction

This chapter explains the motivation for this work, states its objectives and gives an overview of the document structure.

1.1 Motivation

Model-based user interface development is an approach which aims at decreasing the effort needed to develop user interfaces (UIs) [MPV11]. But in order to be practicable, model-driven software development requires adequate tool support. At the current state of the art, automated generation especially of graphical UIs (GUIs) based on an interaction model of some kind is possible. But while many approaches to model-based GUI development offer some kind of tool support, most of them lack tool support that covers the whole development process.

The Unified Communication Platform (UCP) is a specific toolset for developing GUIs in an iterative and incremental fashion [PRK13]. Those user interfaces are based upon an interaction design which is composed from several model types specified in UCP. Creating such an interaction design typically involves designing alternatives, prototyping and evaluating, and these activities are to be repeated for informing each other [PRS11]. Unlike some related approaches, UCP does not assume the availability of a complete and final model of the interaction design. Instead, it defines an iterative and incremental process for developing and gradually improving the interaction design for the GUI. This process utilizes automated generation of GUIs in order to decrease the time that is required to build a prototype of the GUI. This in turn makes it easier and quicker to get feedback for refining and extending the interaction design in further iterations.

The existing tool support in UCP already covers many important parts of the process. It features graphical editors for the different model types which are required by the process. Verification facilities provided by UCP help discovering and fixing errors in the discourse model in early stages of an iteration. Graphical user interfaces can be automatically generated for different device types. The generated user interfaces can then be customized to some degree using graphical tools, which again allows to easily evaluate alternatives.

But there is also some room for improvements. In its current form, the tool support in UCP does not cover the whole GUI development process. Several important aspects still have to be done manually, which is error-prone and time-consuming. Also, the development process is not really intuitive and requires "too much" detailed knowledge about some internals of UCP, even for basic scenarios, from the developer.

Just starting a new project from scratch is one of the particularly tedious tasks, yet with the current state of the UCP tools it has to be performed manually for every new project and by every novice GUI developer. The model files which are required for a new project need to be created individually and linked together appropriately. Some of the default values need to be changed, or else they will cause the automated GUI generation to fail later on. Also, the GUI generation tools require a very specific structure to be present in the interaction models in order to work correctly, which also needs to be created manually. If any of these points is omitted, GUI generation will fail.

Such repetitive actions could be easily avoided by more automation. If there were a kind of template for a new project in UCP, or some kind of "New Project" wizard that sets up all those prerequisites automatically, all the above critical steps would become obsolete or could at least be significantly simplified. This would not only save costly GUI development time, but also remove many possible error causes.

Also the usability of the existing UCP tools could be improved. When the automatic GUI generation fails, the reason is often not clearly indicated to the GUI developer. Making error messages more descriptive would make it easier to track down the cause of the error in such cases. Improving the model verification tools would allow certain kinds of errors to be noticed even earlier.

While UCP provides automated generation of the GUI code and even a runtime environment for the generated GUI, the Application Adapter (which is an important part of the application back-end that is needed to feed some sample data to the GUI) currently needs to be hand-crafted. This involves manually creating code that handles every single Communicative Act specified in the Discourse Model. Again, this task could be significantly simplified if an application back-end stub could be generated automatically using the information which is already specified in the interaction model.

Existing shortcomings of the UCP tools make it harder especially for new GUI developers to get started using the platform. Improving the tool support would speed up the time required to get started with UCP, eliminate many possible sources of errors and increase productivity. Better usability of the UCP tools would reduce the time that is required to walk through any project. Also, by attempting to make the tool support more self-explanatory, the amount of details that a GUI developer needs to learn about UCP in order to be able to get a project done would be reduced.

1.2 Objective and Approach

The major objective of this work was to improve the tool support in UCP in order to make the process more intuitive, less time-consuming and easier to learn for the GUI developer. This has been addressed by adding several new tools to UCP to automate some additional aspects of the process and by improving some of the existing tools. Ultimately, new UCP users should be able to get started developing GUIs with UCP more quickly.

1.3 Definitions

There are no officially released versions of the UCP tools that reflect their state before and after this work which this document could refer to. For the sake of better readability, the state of the

tool support in UCP before this work will be referred to as "version 1" in the context of this document, and the state of the UCP tools after this work as "version 2".

1.4 Structure of the Document

The remainder of this work is structured as follows:

- Chapter 2 – Background and Related Work – provides an overview of the existing tool support in UCP, addresses its shortcomings and gives a brief comparison with related work.
- Chapter 3 – New Tools – discusses the tools that have been newly introduced to UCP in the context of this work.
- Chapter 4 – Improvements to Existing Tools – describes the improvements made by the author to the existing tools in UCP.
- Chapter 5 – Evaluation – compares the improved UCP tools to the tool support in version 1.
- Chapter 6 – Discussion and Outlook – reviews the results of the work and provides some outlook.
- Chapter 7 – Conclusion – provides the conclusions and highlights some issues that may be worth of further investigation.

2 Background and Related Work

This chapter provides information about the concepts behind UCP and its tool support in version 1 which the other chapters in this work refer to. For comparison, it also briefly outlines the tool support in related work.

2.1 UCP Overview

UCP facilitates model-based development of GUIs in an iterative and incremental fashion. It specifies an underlying interaction design for such GUIs and provides tools for editing it. UCP also features automatic generation of GUIs out of the models representing this interaction design.

The following two sections outline the concepts of the interaction design in UCP and describe the GUI development process. Then, an overview of the tool support in version 1 of the UCP tools and an example how they can be used to develop a GUI for a new "discourse project" will be given.

2.1.1 Interaction Design

In UCP, the high-level communicative interaction design between two parties (called "agents") is specified in the form of three model types which are related to each other: a Discourse Model, a Domain-of-Discourse Model and an Action-Notification Model [PRK13].

The Discourse Model specifies the interaction using so-called Communicative Acts (such as `OpenQuestion`, `Answer`, `Request` or `Accept`). Communicative Acts are the basic building blocks of Discourse Models in UCP. Adjacency Pairs, which relate one opening and zero to two closing Communicative Acts, model typical turn-takings in a conversation (e.g., question–answer or offer–accept). The Adjacency Pairs are linked together in a hierarchical way by Rhetorical Structure Theory (RST) relations (e.g., `Alternative`) or Procedural Constructs (such as `Sequence` or `Condition`) to model more complex interactions. For example, a `Joint` relation links two or more Adjacency Pairs that may be executed concurrently, while the Adjacency Pairs linked to a `Sequence` must be executed in the order specified in the Discourse Model. An `IfUntil` is an example for an even more complex Discourse Relation, which models a loop which is executed as long as an associated condition is fulfilled.

Figure 2.1 shows a small example for a Discourse Model. In that graphical representation, Adjacency Pairs are depicted as diamond-shaped elements and Communicative Acts as rectangular shapes with rounded corners. The green/yellow color of a Communicative Act shows which agent it is associated with ("user" or "system"). In this example, the left sub-branch of the `Sequence` Discourse Relation will be executed first. The "system" will ask the "user" to pick an item from a list of possibilities, which is modeled as an opening Communicative Act of type `ClosedQuestion`, marked as (1). The user will then provide an answer, which is modeled as the associated closing Communicative Act (2). Then, following the second sub-branch of the `Sequence` element on the right side of the diagram, the system will make an offer to the user (3), which the user then can either accept (4) or reject (5).

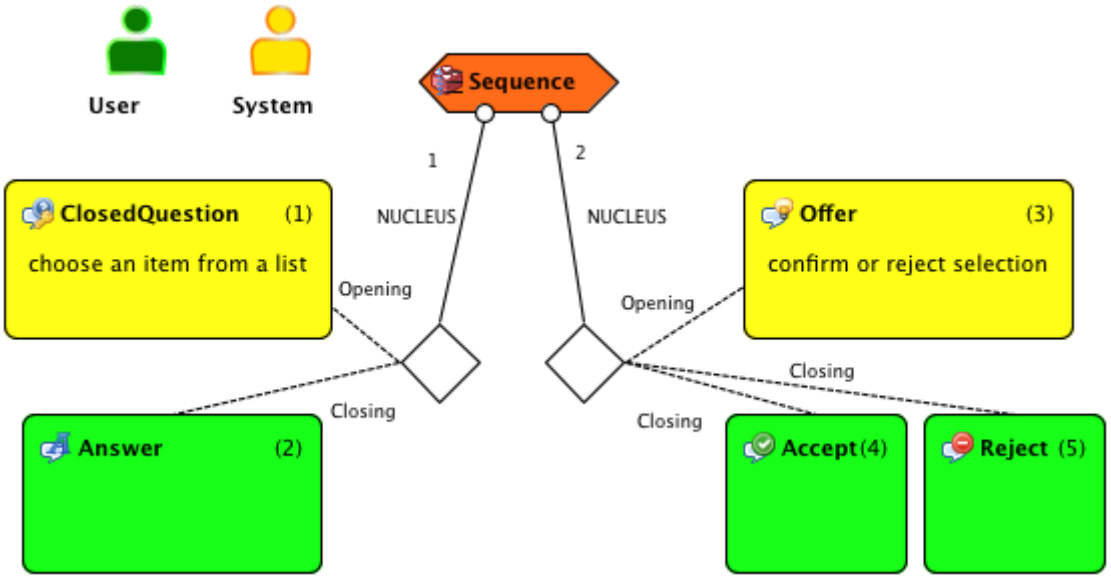

Figure 2.1: An Example Discourse Model.

A Discourse Model refers to a Domain-of-Discourse Model which specifies the concepts that the two interacting agents can "talk about". The connection between the Discourse and the Domain-of-Discourse Model is established through the propositional content specified for a Communicative Act. In addition, the propositional content refers to the Action-Notification Model, which specifies the actions and notification to be performed by the interacting parties (see [Pop12] and [PR11] for more details).

2.1.2 Development Process

The GUI development process in UCP consists of several different activities [RKP+14]. Figure 2.2 shows a stripped-down version of that process in Business Process Model and Notation (BPMN[1]). It focuses on those activities which are required for creating a runnable GUI (prototype) for a new project using UCP. The process described in [RKP+14] additionally covers the iterative and incremental aspects of the GUI development process in UCP. Those can be considered optional in the context of this work since they do not introduce the need for any additional tools other than those which are required for completing the activities shown in Figure 2.2.

[1] http://www.omg.org/spec/BPMN/2.0

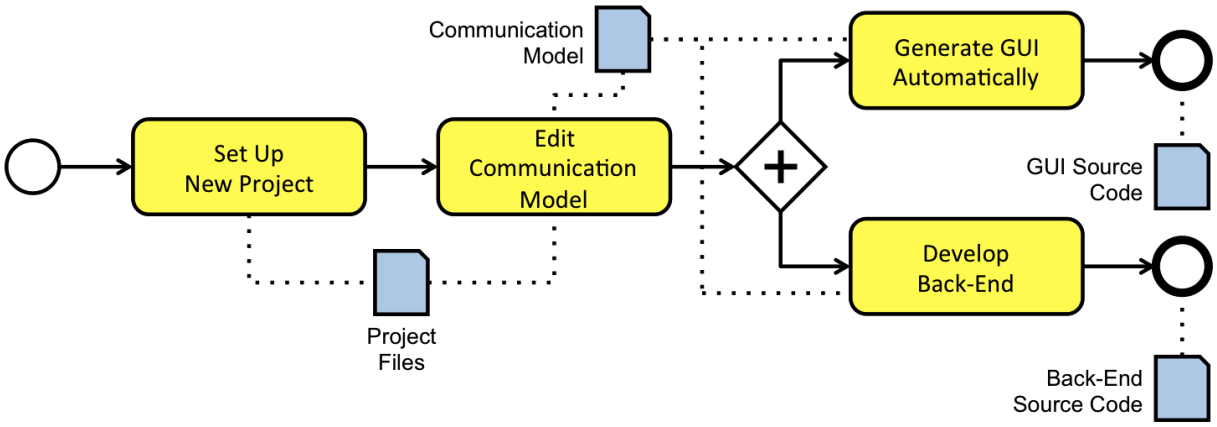

Figure 2.2: GUI Development Process in UCP.

The activity *Set Up New Project* involves the initial creation of a new project and the required model files (the Discourse, Domain-of-Discourse and Action-Notification Models). This activity only needs to be performed once for each project. The actual communication model is created in the course of the *Edit Communication Model* activity by editing those model files. Based on that communication model, the UCP tools can automatically generate an (intermediate) user interface model (called "Screen Model") and the GUI source code as result of the *Generate GUI Automatically* activity. The Screen Model is a toolkit-agnostic representation of the GUI. It allows the communication model to be evaluated and refined independently of the UI technology (e.g., HTML) that will be used for the final GUI. The characteristics of the target device are passed as parameters to the GUI generation process in the form of a so-called Device Specification. The generated GUI can be further customized by specifying custom rendering rules in a Rendering Rules Model. More details about the automatic GUI generation process (and the Screen and Rendering Rules Models, as well as the Device Specifications) are available in [RPK+11]. Finally, in order to make the generated GUI work, a back-end that feeds some sample data to the GUI needs to be developed in the activity *Develop Back-End*. The GUI source code and back-end source code together form a runnable version of the generated GUI.

Both the intermediate Screen Model and the final GUI can be evaluated by the GUI developer (or other subjects). As a result of such an evaluation, a step back to refining the interaction model can be taken, which results in a further iteration. The UCP process supports both micro-iterations (when the new iteration is a result of evaluating the Screen Model) and macro-iterations (as a result of evaluating the final GUI). In each iteration, after the communication model has been adapted in the *Edit Communication Model* activity, the activities *Generate GUI Automatically* and *Develop Back-End* need to be repeated.

2.1.3 Existing Tool Support

The tool support in UCP is built upon the Eclipse[2] platform. Eclipse is a feature-rich Integrated Development Environment (IDE) that was originally oriented towards Java development. An Eclipse installation can be extended by plug-ins which add new features to the IDE. For example,

[2]http://www.eclipse.org/

support for many different programming languages and frameworks exists in the form of plug-ins that can be added to Eclipse.

The Eclipse platform provides a broad range of Application Programming Interfaces (APIs) that can be consumed by such plug-ins. This allows custom plug-ins to gain access to the Eclipse workspace, create or modify projects and files, add new views to the Eclipse IDE and customize the menu structure, among other things. Of particular importance for the UCP tools are the Eclipse Modeling Framework[3] (EMF) and its close relative, the Graphical Modeling Framework[4] (GMF). EMF adds support for model-based software development to Eclipse. It provides a core meta-model (Ecore) that can be used to express other models. It also features automatic code generation for such custom models and a basic editor. On top of EMF, GMF provides support for automatically creating code for graphical editors for other model types. Using GMF, it is easy to create and customize a graphical editor for a custom EMF model.

UCP uses such an Ecore model to specify the meta-model of its interaction models. Based on that meta-model, it features EMF editors for the resulting model types (the Discourse, Domain-of-Discourse and Action-Notification Models). But more importantly, it provides a customized graphical editor (based on GMF) for those models.

At the heart of the UCP tools is the graphical Discourse Model editor. It allows editing the interaction model using a point-and-click user experience. In this graphical editor, different kinds of model elements (such as Discourse Relations, Communicative Acts etc.) are represented by different visual elements. Figure 2.1 shows a screenshot of a Discourse Model in the graphical editor. In contrast, the basic EMF editor only provides a tree view of the Discourse Model. While the EMF editor is sufficient for editing the Discourse Model, the graphical variant is much more convenient to use. The UCP tools also feature an editor for the Action-Notification Model, the Screen Model and the Rendering Rules Model. For the Domain-of-Discourse Model a plain Ecore model is used (for which Eclipse provides a graphical editor).

Both EMF and GMF support model verification through constraints. Such constraints can be expressed in various forms, e.g., as Object Constraint Language (OCL) expressions or Java code. Eclipse features both constraints that are evaluated continuously while a model is being edited, as well as constraints that are evaluated when the developer manually selects a menu item. The UCP tools make use of constraints to specify additional restrictions for its model types that are not represented in the meta-model.

For automatically generating the GUI from the interaction model, the UCP tools provide a custom type of Eclipse launch configuration. It features a visual editor for specifying all generation parameters, which is shown in Figure 2.3. The values entered by the GUI developer are stored to a file which makes re-generation using the same settings easy. When such a launch configuration has been created and configured properly, the actual GUI generation process can be easily invoked through a menu item in the Eclipse IDE. The GUI generation process results in two new Eclipse projects being created. One contains a device-agnostic Behavioral UI Model, the other one a device-specific Structural UI Model, as well as the actual GUI code. Both of these model types can be edited by the GUI developer using editors provided by the UCP tools.

The UCP tools also provide a runtime environment for the generated GUI. This runtime is powered by a state machine for the different GUI screens, which is also generated automatically as part of the GUI generation process. For serving the HTML pages of the individual GUI

[3] http://www.eclipse.org/modeling/emf/
[4] http://www.eclipse.org/modeling/gmp/

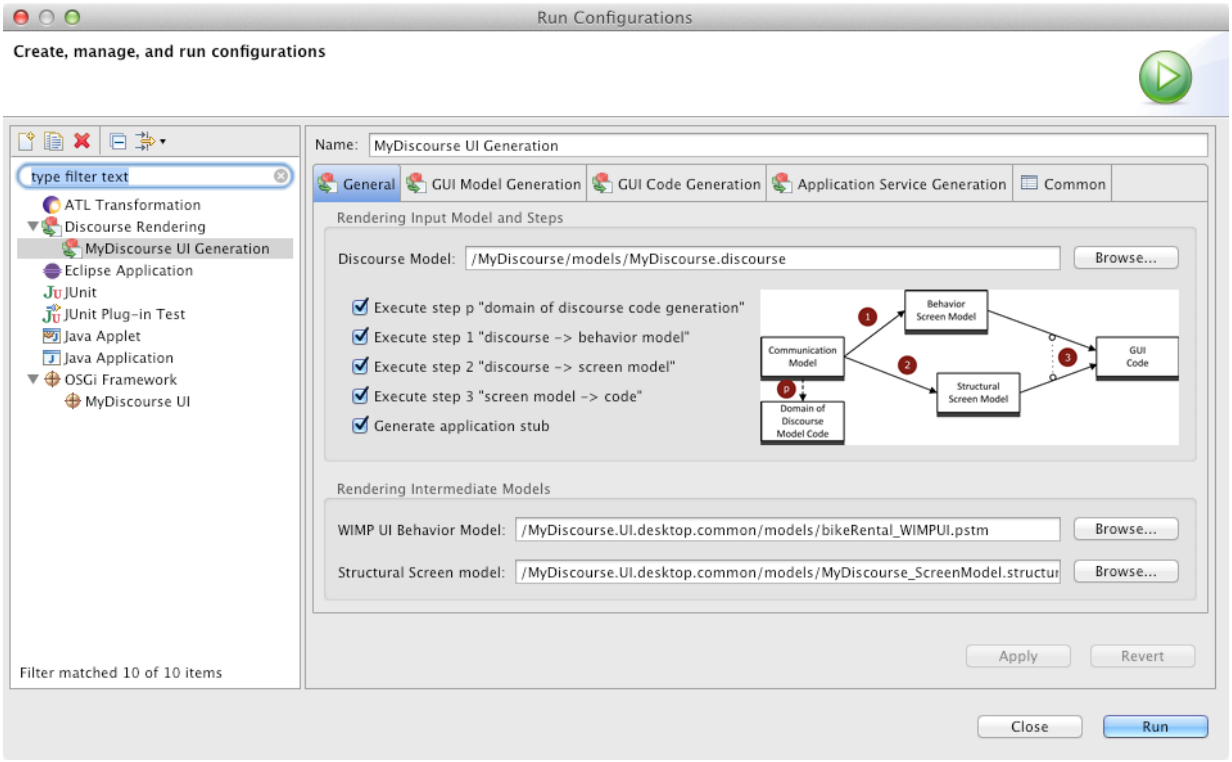

Figure 2.3: The GUI Generation Launch Configuration Editor.

screens, an embedded Web server (based on Jetty[5]) is used. For providing the input of the other interaction party (represented by the "system" agent), the UCP runtime makes use of a so-called Application Adapter. It is an implementation of a message-based interface specified by UCP, which processes individual Communicative Acts in the course of an interaction. While the GUI code (as well as the underlying state machine as part of the UCP runtime) are generated automatically by the UCP tools, in version 1 of the UCP tools the Application Adapter needs to be created manually. Similar to the GUI generation process, the UCP runtime can be configured and started using a custom Eclipse launch configuration.

2.1.4 Sample Project Walkthrough

This section provides a walkthrough of a new discourse project using version 1 of the UCP tools. It demonstrates how the tool support in UCP can be used to realize a discourse project, describing every individual step that has to be performed manually by the GUI developer.

When a new discourse project is started in version 1 of the UCP tools, the following steps are necessary in order to create a runnable GUI prototype:

1. Create a new Java project in the Eclipse workspace.

2. Create a new Discourse Model in that project.

[5]http://www.eclipse.org/jetty/

3. Create a new Ecore model file that is used as the Domain-of-Discourse Model for the discourse project.
4. Create a new Action-Notification Model.
5. Add a reference to the Domain-of-Discourse Model to the Discourse Model.
6. Add a reference to the Action-Notification Model to the Discourse Model.
7. Change the `name` property of the Discourse Model to a value containing only letters and digits.[6]
8. Create an action for starting the interaction in the Action-Notification Model.
9. Create the inserted sequence necessary for starting the interaction in the Discourse Model.
10. Create some domain objects in the Domain-of-Discourse Model.
11. Create some Communicative Acts in the Discourse Model using the graphical editor.
12. Copy the Device Specifications to the Eclipse workspace, if they don't already exist.
13. Create a launch configuration for the GUI generation and fill in the right values.
14. Perform GUI code generation by running the launch configuration.
15. Create a new project for the Application Adapter using the "New Service Project" wizard.
16. Create an implementation of the Application Adapter that handles every action and notification specified in the Discourse Model (this involves revisiting each Communicative Act in the Discourse Model and creating an appropriate code branch).
17. Create a launch configuration for starting the UCP runtime and add all required projects (the Application Adapter, as well as the two generated GUI projects).
18. Start the UCP runtime using the launch configuration in the service project.

Steps 1 to 9 are about setting up a new discourse project. They correspond to the activity *Set Up New Project* in Figure 2.2. These steps have to be performed for each new discourse project and result in a project structure that contains all files required for GUI generation. The same result can be achieved by simply copying an existing project, but that requires a lot of manual changes to be done afterwards. In both scenarios, the project name has to be changed in various places (also within the model files), which is error-prone because it is easy to overlook some occurrences. Chapter 3 introduces a wizard for setting up a new discourse project, which automates these nine steps.

Steps 10 and 11 are about editing the models representing the interaction design. They correspond to the activity *Edit Communication Model* in Figure 2.2. This is the actual, creative work that is related to the use cases of the application being developed and can, therefore, hardly be automated. Still, improvements (mainly usability-related) have been done to the involved editors, which are covered in Chapter 4.

[6] This value is used as part of a Java identifier during GUI code generation. In version 1 of the UCP tools, the default value ("New Discourse") contains a space character which results in compilation errors in the generated code.

The prerequisites for automatic GUI generation are prepared in Steps 12 and 13. The generation process requires access to the Device Specifications, which need to be copied to the Eclipse workspace. Preparing the launch configuration for the GUI generation is a particularly tricky step because it requires multiple adjustments of manual settings by the GUI developer (paths to model files, names for the generated output artifacts and other generation parameters). Even if this file is copied from another project, it is still most likely that many property values have to be changed. The actual GUI generation is then performed in Step 14. This corresponds to the activity *Generate GUI Automatically* in Figure 2.2.

Steps 15 to 17 cover the creation of an Application Adapter for the UCP runtime. They correspond to the activity *Develop Back-End* in Figure 2.2. While version 1 of the UCP tools features a wizard for creating an Eclipse project for the Application Adapter, the actual implementation needs to be created manually. This includes the creation of a launch configuration for starting the UCP runtime. In this launch configuration, the required plug-ins and their `auto-start` values need to be specified manually. Finally, the UCP runtime is started in Step 18, which is a prerequisite for accessing the GUI in a Web browser.

After all these steps have been performed, the GUI of the discourse project can be accessed using any browser on desktop as well as mobile devices.

The list only contains the essential steps that are required to create a runnable prototype of the GUI. Since the UCP process is iterative, it is likely that the interaction model will be refined in some further iterations. After such a modification, the GUI generation process needs to be repeated and possibly also the Application Adapter will have to be adapted according to the changes in the interaction model.

There are also some optional steps likely to be involved when evaluating the generated GUI. For example, the UCP tools provide the GUI developer with a possibility to customize the intermediate Screen Model and specify some custom rendering rules in a Rendering Model file. These customizations are, however, not mandatory for a discourse project.

2.2 Related Work

This section outlines the tool support available in some related work. More detailed information can be found in [Kan14].

2.2.1 MARIA

The Model-based Language For Interactive Applications (MARIA) is a task-based approach for interaction modeling, based on the Cameleon Reference Framework[7] [PSS09] [PSS11]. The tool support in MARIA consists of two environments, the ConcurTaskTrees Environment (CTTE) and the MARIA Environment (MARIAE).[8]

CTTE features a graphical editor for the task-based interaction model (called "Task Model" in MARIA) [Pat03]. Figure 2.4 shows the Task Model of an example project in the CTTE editor. Similarly to the graphical Discourse Model editor in UCP, this editor features model verification

[7] http://giove.isti.cnr.it/projects/cameleon/deliverable1_1.html
[8] http://giove.isti.cnr.it/tools/MARIA/home

based on compliance checks with a meta-model. In addition, CTTE contains a simulator that can be used to validate the interaction model, shown in Figure 2.5. In this simulator, certain conditions associated with tasks (e.g., whether the user has sufficient privileges) can be entered manually. This allows the interaction designer to investigate different paths of the interaction flow.

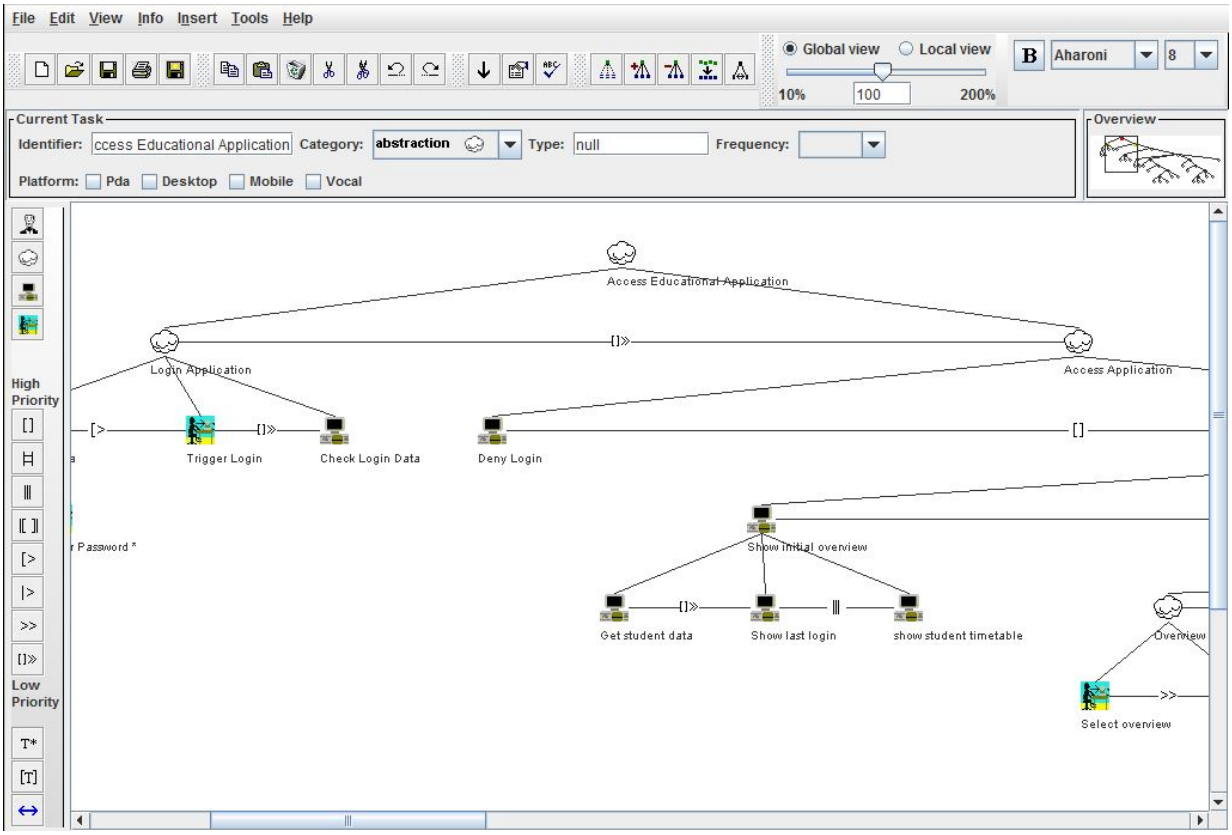

Figure 2.4: The ConcurTaskTrees Environment.

After such a Task Model has been created in CTTE, a GUI can be created and connected to a back-end service using MARIAE. MARIAE contains some predefined transformations for supported output platforms (Web browsers, Smartphones and Vocal UIs). These transformations cannot be customized by the GUI developer (in contrast to UCP, where the output of the GUI generation process can be customized for different device types using Device Specifications). This allows for rapid prototyping, because it is easy to generate a runnable version of the GUI from a Task Model. Figure 2.6 shows the Task Model from the previous example in MARIAE.

MARIAE supports different kinds of Web services (REST and SOAP[9]) for the application back-end of the generated GUI. It features a graphical editor for specifying the mapping between tasks from the Task Model and requests to the Web service. This editor can, e.g., load the WSDL description of a service. Based on it, MARIE allows the GUI developer to associate tasks with the operations supported by the service. MARIAE assumes the availability of a back-end service for the application, it does not include any support for developing such a service. The Web service which provides the back-end for the GUI to be developed, therefore, needs to be

[9] http://www.w3.org/TR/soap12-part0/

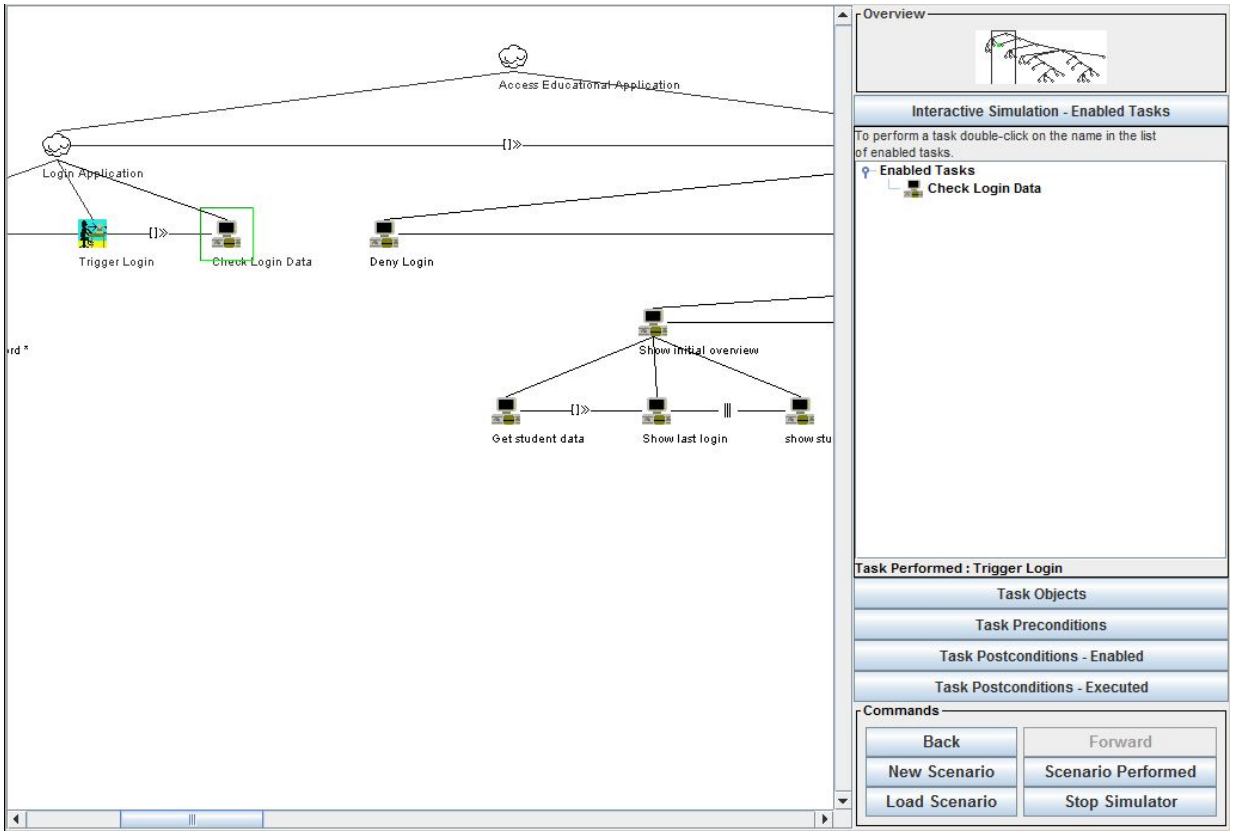

Figure 2.5: Simulator for Task Models in CTTE.

provided externally. In contrast, developing a stub for the application back-end is part of the development process in UCP.

When the mapping is defined, the GUI can be opened using, e.g., a Web browser on a desktop or mobile device. MARIAE features a graphical editor that allows the GUI developer to customize the structure of the underlying GUI model, which is shown in Figure 2.7. This does not, however, modify the final GUI code, but rather the UI model. The GUI code needs to be re-created afterwards.

Although the simulator in CTTE allows some early evaluation of the Task Model, MARIA does not support iterative development of the interaction design. After changes have been made to the Task Model in CTTE, any adaptations of the GUI have to be discarded and the GUI developer has to start with MARIAE from scratch.

In comparison with the UCP tools, which come in form of one IDE, the MARIA tools require the GUI developer to switch between multiple development environments, which adds some undesired overhead to the development process. Both UCP and MARIA feature graphical editors and model verification, but UCP lacks a tool for model validation through simulation, as provided by CTTE. A GUI can be generated automatically in both UCP and MARIA, but the UCP tools allow for more customization with respect to different target device types. Developing the back-end for the generated GUI is out of scope for MARIA, but it is part of the development process in UCP and tool support for this activity has been added in version 2 of the UCP tools. Furthermore, the UCP process (and the UCP tools) facilitate both iterative and incremental GUI development, which is not supported by MARIA.

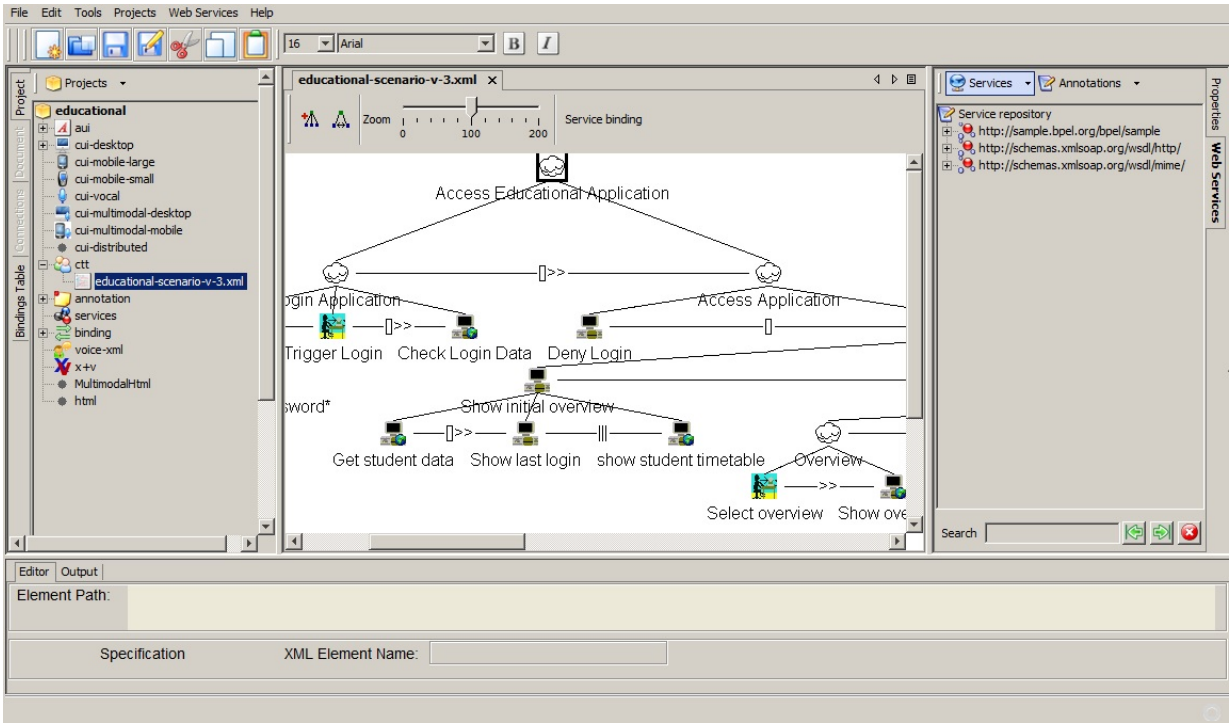

Figure 2.6: The MARIA Environment.

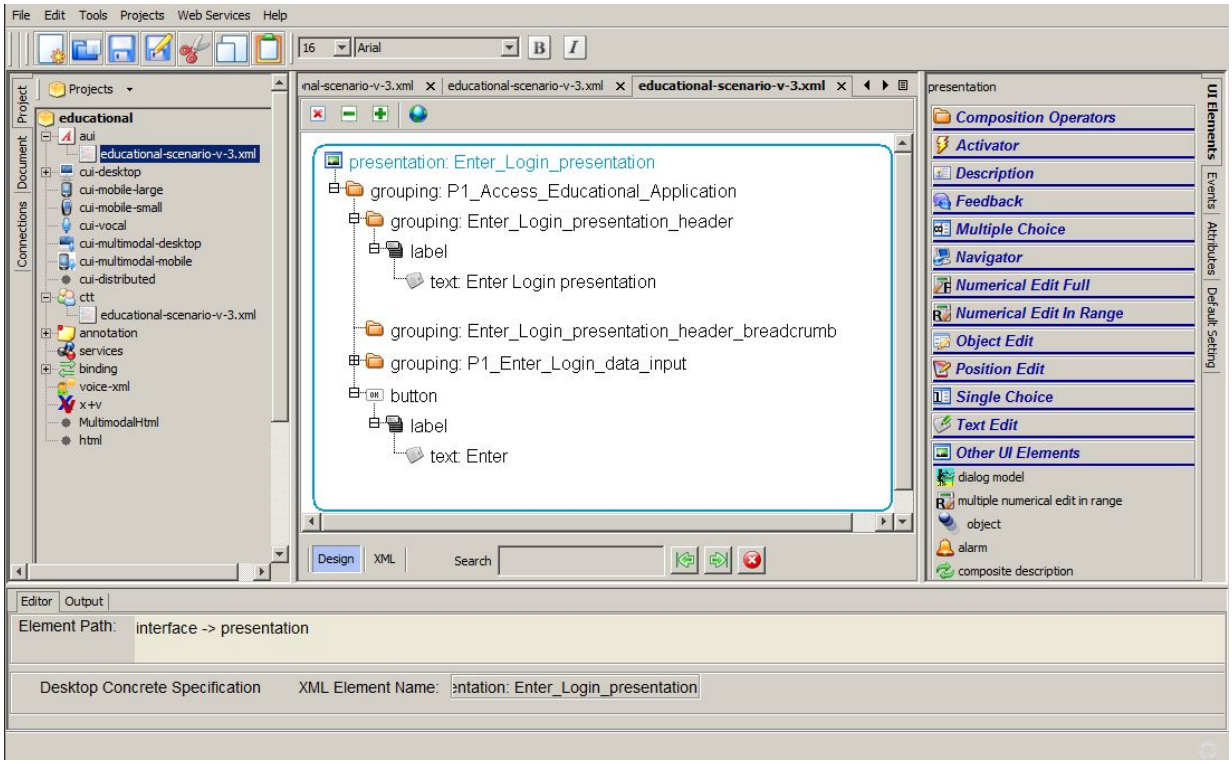

Figure 2.7: Customizing the Concrete User Interface in MARIAE.

2.2.2 UsiXML

The User Interface Extensible Markup Language (UsiXML[10]) is an XML-compliant markup language that describes the UIs for multiple contexts of use such as Character User Interfaces (CUIs), Graphical User Interfaces (GUIs), Auditory User Interfaces and Multimodal User Interfaces [LVM+04]. It is also based on the Cameleon Reference Framework and provides meta-models on all four levels of abstraction described in this framework. UsiXML consists of a User Interface Description Language (UIDL), which describes various aspects of a user interface under development, independently of its physical characteristics. UsiXML aims at providing a full model set which is capable of covering every aspect of an interaction on different levels of abstraction.

UsiXML does not offer any publicly available tools that could be evaluated and compared to the tool support in UCP. While the documentation indicates that some proprietary graphical editors might exist for at least some of the models, it is unclear how these tools can be obtained. The GUI developer, therefore, needs to create and edit the XML files manually using XML editors. Also, the conversion between different model types (and abstraction levels) needs to be performed manually.

UsiXML focuses on the interaction design and does not provide means for creating a working prototype using any concrete GUI technology. The GUI developer may himself be able to create tools for automatically generating a runnable GUI prototype from some UsiXML interaction models, but no support for such an activity is explicitly provided by UsiXML.

Since no tools for UsiXML could be obtained, no direct comparison with the tool support in UCP is possible.

[10] http://www.usixml.org

3 New Tools

This chapter describes in detail the new tools that have been added in version 2 of the tool support in UCP by the author of this work. The goal was to cover certain aspects of the GUI development process that were lacking tool support in version 1 of the UCP tools.

3.1 "New Discourse Project" Wizard

Every UCP project consists of a certain set of different types of files (model files, rendering rules, launch configurations, etc.) Whenever a new project is started, those files have to be created and linked together appropriately. In version 1 of the UCP tools, every required file had to be created manually. The "New Discourse Project" wizard is a new tool that was introduced in version 2 in order to automate this process as much as possible.

3.1.1 Implementation as an Eclipse Wizard

The original idea was to guide the user through the project setup with the help of an Eclipse "Cheat Sheet". Such Cheat Sheets are designed to guide the user through a series of steps in order to complete a task [CR06, Chapter 15.5]. If required, they can even launch some tools automatically. The cheat sheet for setting up a new discourse project would then appear to the user as some kind of check list that reminds him which step he needs to perform next. That would be an improvement since the user would no longer have to remember which files he needs to create and which other actions he needs to perform. But the major drawback of such an approach would be that the user is still required to do many individual steps manually.

Instead, it was chosen to implement a "New Discourse Project" wizard. This solution allows a greater amount of work to be done automatically while requiring less user interaction. Instead of guiding the user through multiple individual steps, all required actions are combined in one wizard that collects the required data from the GUI developer and then performs the project setup automatically. Also, Eclipse users are already familiar with the `New` → `Project...` menu for setting up various kinds of projects (e.g., Java projects). Including the "New Discourse Project" wizard in that menu makes it easy to find for novice users.

The wizard for creating new discourse projects is implemented in a new Eclipse plug-in that was added to the UCP tools in this work, `org.ontoucp.workbench.discourse.wizard`. The wizard

itself is defined in the `NewDiscourseWizard` class, which implements the `INewWizard` interface [CR06, Chapter 11.2]. To make the wizard appear in the `New Project` dialog in Eclipse, it has to be registered in the `plugin.xml` of the plug-in. This is done by adding the extension definition from Listing 3.1 to that file.

```
<extension point="org.eclipse.ui.newWizards">
   <category
      id="org.ontoucp.workbench.discourse.wizard.wizards"
      name="OntoUCP">
   </category>
   <wizard
      category="org.ontoucp.workbench.discourse.wizard.wizards"
      class="org.ontoucp.workbench.discourse.wizard.NewDiscourseWizard"
      finalPerspective="org.ontoucp.workbench.perspective"
      icon="icons/DiscourseEditor16.png"
      id="org.ontoucp.workbench.discourse.wizard.NewDiscourseWizard"
      name="Discourse Project"
      project="true">
      <description>
         Creates a new discourse project.
      </description>
   </wizard>
</extension>
```

Listing 3.1: "New Project Wizard" Extension Definition.

When the new plug-in is loaded in Eclipse, a new category "OntoUCP" appears in the `New Project` wizard that contains the "Discourse Project" project type. The custom wizard can be promoted to the top of the list by marking it as a "primary wizard". This is achieved with the following addition to the wizard definition:

```
<extension point="org.eclipse.ui.newWizards">
   ...
   <primaryWizard
      id="org.ontoucp.workbench.discourse.wizard.NewDiscourseWizard">
   </primaryWizard>
</extension>
```

Listing 3.2: Primary Wizard Extension Definition.

Figure 3.1 shows the screen that appears when the user chooses the `New` → `Project...` menu item. It displays the list of project types supported by the Eclipse installation, which contains the "OntoUCP" category and the "Discourse Project" wizard at the very top of the list.

An Eclipse wizard provides data input capabilities that can span multiple pages which the user can navigate through using "Back" and "Forward" buttons. It also has a "Cancel" button that can be used to abort the operation and a "Finish" button which triggers a certain action that processes the input from the wizard pages.

The wizard for creating new discourse projects requires only two input elements, as shown in Figure 3.2. One is the name of the discourse project. The other one is the Java package name to be used for the generated code. It was attempted to keep the amount of information that the

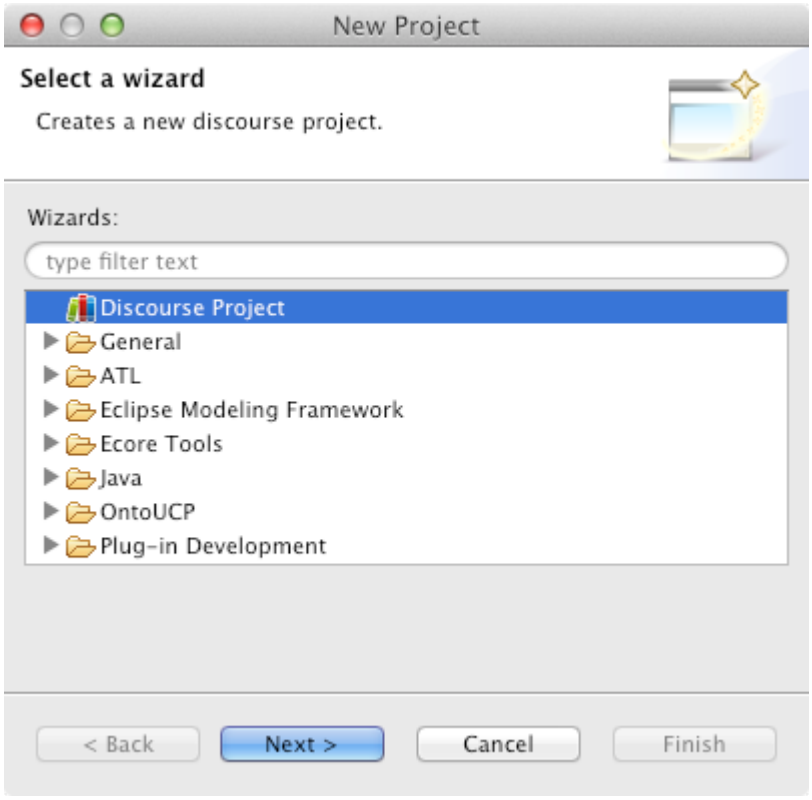

Figure 3.1: Project Type Selection Screen.

user needs to enter at a necessary minimum in order to make creation of new discourse projects as easy as possible.

This wizard page (which hosts the input fields and labels) is implemented in the `ProjectPage` class. This class is also responsible for verifying the input, making sure that there is no other project with the specified name and that the value provided for the package name is a valid Java namespace identifier. The "Finish" button is disabled as long as the values provided by the user are not correct. For example, Figure 3.3 shows the wizard as it appears when the chosen project name is already in use by another project in the workspace.

3.1.2 Automatic Project Setup

Once the user has filled in the project and package names and clicked the "Finish" button on the wizard, a project with the specified name is generated in the Eclipse workspace. The logic for this operation is also contained in the `NewDiscourseWizard` class, but makes use of several auxiliary classes to do the actual work. Using the API provided by the Eclipse platform, the `JavaProjectCreator` class creates a new Java project with the name specified in the wizard. It also sets up the Java classpath for the new project and creates the `src` and `bin` folders. The `DiscourseProjectConfigurator` class then generates all model files which are required for an UCP project in the project:

- The Domain-of-Discourse Model (and the accompanying diagram file for the graphical GMF editor).

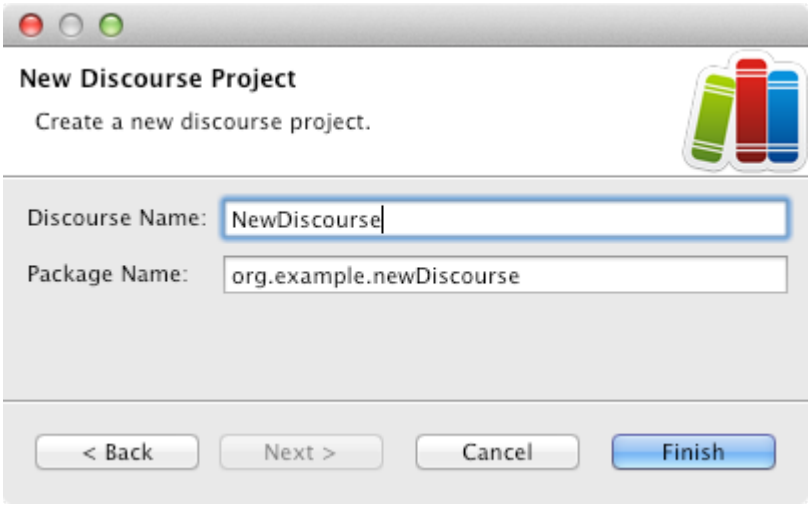

Figure 3.2: The "New Discourse Project" Wizard.

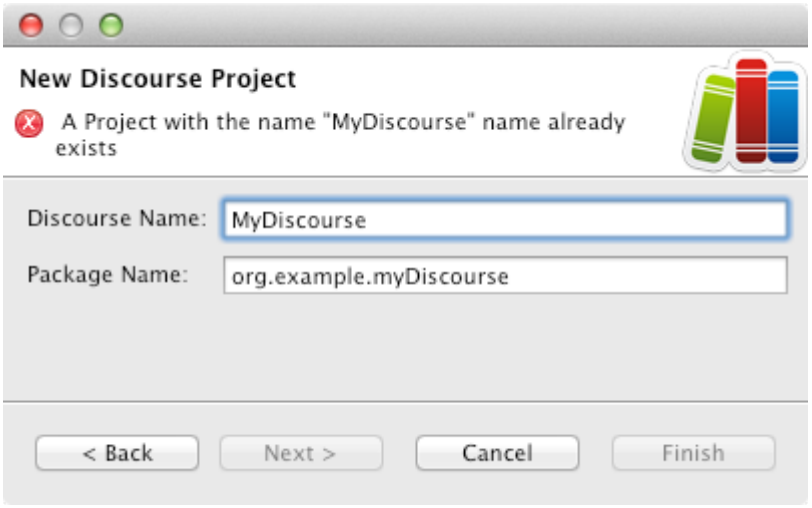

Figure 3.3: The "New Discourse Project" Wizard Showing an Error.

- The Action-Notification Model.
- The Discourse Model (and the accompanying diagram file for the graphical Discourse Model editor).

The file contents are created using Java Emitter Templates (JET).[1] Templates for each generated file are contained in the `org.ontoucp.workbench.discourse.wizard` plug-in bundle. These templates contain placeholders for the project and package names which are replaced with the values provided in the wizard when the template is evaluated.

The model file templates do not simply yield empty model files, the generated files have a few characteristics that are set up in advance:

- The Discourse Model contains links to the generated Action-Notification Model and Domain-of-Discourse Model.

[1] http://www.eclipse.org/emft/projects/jet/

- The Action-Notification Model contains an action that represents the user's wish to start an interaction.
- The Discourse Model also contains the inserted sequence which is required to start the interaction.[2]

This eliminates the need to set these things up manually. The GUI developer cannot forget any of these steps anymore and can additionally save some time. This is potentially a big improvement since some kinds of errors (e.g., the Action-Notification Model not being referenced by the Discourse Model) are hard to spot.

The implementation in the `DiscourseProjectConfigurator` class also creates the `MANIFEST.MF` file for the new project, as well as the `build.properties` file. JET templates are used to generate these files as well. The build properties are set up to contain the directory containing the model files in the binary distribution, as shown in Listing 3.3:

```
source.. = src/
output.. = bin/
bin.includes = META-INF/,\
               .,\
               models/
```

Listing 3.3: Build Properties for the Generated Project.

The "New Discourse Project" wizard also adds a default Rendering Rules file to the generated project, which can then be extended or modified by the GUI developer. Figure 3.4 shows all the files of a project generated by the wizard.

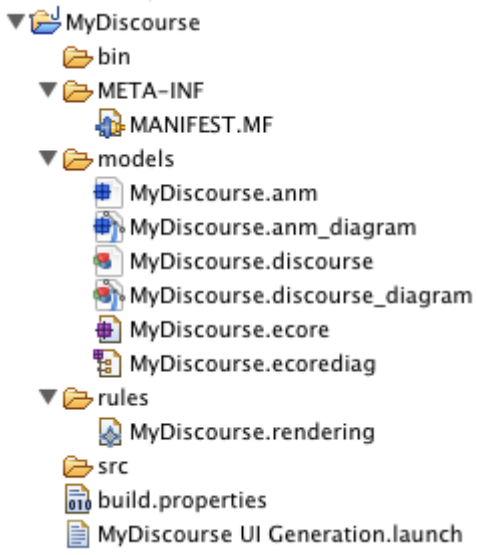

Figure 3.4: Project Generated by the "New Discourse Project" Wizard.

[2]Because the UCP runtime is implemented as a Web service, the interaction needs to be initiated by a service request from the client (which happens when the user opens the page in a Web browser). The Discourse Model needs to contain a Communicative Act that represents this action, which is modeled in the form of an "inserted sequence".

3.1.3 Launch Configuration

The "New Discourse Project" wizard does more than just setting up a new discourse project so that the interaction model can be edited. It also prepares all necessary prerequisites for the automated generation of the GUI and an accompanying Application Adapter interface stub (see also Section 3.2).

In UCP, the GUI generation is triggered by a special kind of launch configuration in Eclipse. It consists of a set of properties that can be edited using a graphical launch configuration editor, as described in Section 4.3. These properties contain the names of the output artifacts of the GUI generation, such as the project names and Java namespaces to be used for the generated GUI projects.

The "New Discourse Project" wizard includes such a launch configuration for the GUI generation in the discourse project. This generated launch configuration provides meaningful default values for all properties. The names of the output artifacts are derived from the discourse project name specified in the wizard (e.g., if the discourse project is named "SomeProject", the default value for the generated projects will be "SomeProject.UI.Desktop.common" and "SomeProject.UI.Desktop.html"). The generated launch configuration also links to the generated Rendering Rules file. The wizard furthermore assumes that the Device Specifications are located in the project where the Rich Client Platform (RCP) application puts them by default (see Section 4.2 for more information about the RCP application). It is, therefore, possible to successfully complete the GUI generation process with the default values in the launch configuration as generated by the "New Discourse Project" wizard, without having to change any property. This was an objective for developing the wizard, in order to make GUI generation for new discourse projects as simple as possible. UCP users still have the possibility to modify a wide range of settings in the launch configuration to tweak the generation process as they like.

3.1.4 Constraints

The objective for the development of the "New Discourse Project" wizard was to simplify the creation of a new discourse project. The wizard assumes that it creates a new discourse project from scratch. This assumption allows the implementation of the wizard to be much simpler because it does not need to deal with any kind of file created by a third party. The approach with JET templates works because the set of templates is prepared in advance to be consistent, only the project and package names are replaced with the values provided in the wizard. Creating the missing pieces of a complete discourse project for, e.g., an existing standalone Discourse Model file would require the wizard implementation to be much more sophisticated.

Existing model files could still be used in a new project created by the "New Discourse Project" wizard, but that would most likely require some adaptations (especially renamings) to be done manually.

3.2 Application Adapter Stub Generator

One of the major features of the UCP tools is the ability to automatically generate the GUI code from an interaction model. But UCP does not only generate the front-end code for a given UI

toolkit (e.g., HTML). It also provides a runtime that functions as back-end for the generated GUI code [PRK13].

The UCP runtime is structured using the well-known Model-View-Controller (MVC) architectural pattern [PKR13]. The so-called "Model" contains the core processing and data, the "View" presents output to the user and the "Controller" is responsible for handling the user input. Part of the Model is a state machine which defines the overall behavior of the system in the sense of all possible dialog flows between it and the user. At runtime, the actual (human) user provides the input for the user agent. The Controller then extracts the Communicative Acts associated with the given user input. Afterwards, the state machine in the Model is updated. The Model makes use of a so-called Application Functionality Provider to execute the actions associated with the received Communicative Acts, from the point of view of the system agent. As a result of this, the next GUI screen is displayed to the user.

The Application Functionality Provider is responsible for feeding data to the GUI and also for evaluating conditions that are associated with the system agent. While the remainder of the UCP runtime (including the implementation of the state machine that drives the execution of the application logic) are generated automatically in the course of GUI generation, the actual implementation of the Application Functionality Provider needs to be provided manually in the form of a so-called Application Adapter.

During an interaction, the UCP runtime communicates with the Application Adapter via the `ApplicationAdapterInterface`, which is described in the following section. An implementation of that interface is typically located in a separate Eclipse project (which will be referred to as the "service project" in this document), in order to make it independent of the interaction model and the generated GUI(s). The runtime is started using an OSGi[3] launch configuration, which references the service project, the GUI plug-ins and other plug-ins required by the UCP runtime. Running this launch configuration starts the Eclipse plug-ins of the UCP runtime. After that, the application can be opened in any Web browser.

For the HTML UI toolkit, the UCP runtime makes use of an embedded Web server that serves the individual HTML pages to the Web browser. This allows the back-end for a generated GUI to be started standalone, without the need of an external Web server.

3.2.1 The Application Adapter Interface

The interface for an Application Adapter, `ApplicationAdapterInterface`, is defined in the plug-in `org.ontoucp.discourse.discourseAgent.interfaces`. It is message-based, with the message objects being instances of the `IContent` interface. The `ApplicationAdapterInterface` is defined in the `org.ontoucp.discourse.model.content` plug-in.

The objects implementing `IContent` are, in turn, implementations of either the `IAction` or the `INotification` interface. Objects implementing one of these interfaces are representations of actions or notifications defined in either the `basic` Action-Notification Model, or the custom actions and notifications specified in the Action-Notification Models that are referenced by the Discourse Model. In the Discourse Model itself, actions and notifications are used to specify the content of Communicative Acts, as well as the conditions associated with the child links of certain types of Discourse Relations. They can have objects from the Domain-of-Discourse

[3] http://www.osgi.org/

Model as parameters. The purpose of the Application Adapter is to be able to process each `IContent` message specified in the Discourse Model and to respond with some meaningful data in order to make it possible to navigate through the various screens of the generated GUI. A more detailed description about the overall structure and behavior of the UCP runtime can be found in [PKR13].

The plug-in `org.ontoucp.discourse.discourseAgent` contains an abstract class that can be used as a base class for Application Adapter implementations. This `BasicApplicationAdapter` already implements most of the members of the `ApplicationAdapterInterface`. The most important method on that interface is the `executeAction()` method, which is called by the UCP runtime whenever it requires an input from the system agent (which the implementation of the Application Adapter represents). The `BasicApplicationAdapter` class provides a base implementation of this method which adds some log messages to the console before and after a message has been processed and whenever an error occurs (an exception is thrown). The actual processing of the message is then delegated to the abstract `executeActionInternal()` method, which needs to be implemented by any derived class. The easiest approach to implement an Application Adapter for a discourse project is probably to create a class that extends `BasicApplicationAdapter` and to provide an implementation of the `executeActionInternal()` method. This implementation then needs to return some domain objects (which are generated from the Domain-of-Discourse Model) depending on the type of the action or notification being handled, or `true` or `false` if the `IContent` message represents a condition to be evaluated by the system agent.

The UCP runtime supports multiple simultaneous dialogs with multiple users. Whenever a new user starts a session, a new instance of the Application Adapter is created for that session. Any variables of the Application Adapter implementation are, therefore, related to this session only. In order to be able to maintain an application state that is shared among all sessions, the `ApplicationAdapterInterface` provides the `getApplication()` method. This method returns a shared instance of an application object that is created only once when the UCP runtime is started.

3.2.2 Automatic Generation of the Application Adapter Interface Stub

The UCP runtime requires the following in order to be able to launch the back-end for a GUI:

- An implementation of the Application Adapter interface.
- An "application" class that holds the state which is shared between the individual user sessions (if such a shared state is required for the uses cases of the application).
- A `Service.properties` file that specifies the names of the classes to be used by the UCP runtime for the Application Adapter and the application class.
- The generated code for the domain objects specified in the Domain-of-Discourse Model.[4]
- A launch configuration for starting the UCP runtime.

[4]These types are required by both the generated GUI projects and the Application Adapter. In theory, the Application Adapter could reuse those created in one of the GUI projects during the GUI generation. It was instead chosen to generate a separate copy of these types in the service project. While this introduces some code duplication, it allows for developing the GUI and the Application Adapter independently from each other, even with different development teams, with nothing else than the interaction model in common.

- An Eclipse plug-in that contains all these files (the service project).

Version 1 of the UCP tools already contained a wizard for creating a service project, similar to the "New Discourse Project" wizard. But the project this wizard created was not related to any specific Discourse Model. Instead, it only generated a new project containing empty Application Adapter and application classes. The actual implementation for the `executeActionInternal()` method had to be created manually.

In this work, a new tool was introduced that makes the creation of an Application Adapter much easier. This new Application Adapter interface stub generator is implemented in a separate Eclipse plug-in, `org.ontoucp.service.applicationAdapter.generator`. Just like the service project wizard, it creates a new Eclipse project and generates an implementation of the Application Adapter interface. It also creates a configuration file for the UCP runtime, `Service.properties`. This file specifies (among some other properties) the class name to be used for the Application Adapter interface implementation and for the application class (in the properties `config.ApplicationAdapterInterface.ClassName` and `config.application.ClassName`, respectively). The project setup and file creation is done in a very similar fashion as in the "New Discourse Project" wizard, using the Eclipse platform API and JET templates.

Depending on the type of action or notification that the `IContent` object represents, the Application Adapter needs to execute a different code block or method. The most simple approach is to call `toString()` on the `IContent` instance, which will return the original string in Abstract Syntax Tree (AST) notation, as specified in the Discourse Model. In most of the example projects, a big `switch` statement is then used inside the `executeActionInternal()` method to generate a different response for each message type. Each execution path then handles only one type of message and returns either some domain objects (generated from the Domain-of-Discourse Model) or a `true` or `false` value in the case of a condition evaluation. This `switch` statement, of course, needs to contain the exact same string values as used in the content specifications in the Discourse Model. Creating this code block requires a lot of manual work (typing) and is error-prone because typing errors in the string representations are easy to make but hard to detect. This `switch` is also hard to maintain because each change of a content specification in the Discourse Model requires the corresponding string representation in the `executeActionInternal()` method to be changed as well.

While the "New Service Project" wizard was already able to generate an empty Application Adapter class, the Application Adapter interface stub generator has the additional ability to automatically produce the code for switching between the different types of `IContent` messages. An auxiliary class, `DiscourseCallMapExtractor`, makes use of OCL expressions such as `CommunicativeAct.allInstances()` to scan the Discourse Model for all instances of Communicative Acts and Links. Those OCL expressions are evaluated using API calls provided by the `org.eclipse.ocl.Ecore.OCL` package. Then, the associated actions and notifications are retrieved using methods of the `CommunicativeAct` objects. At this point, the role of the Application Adapter has to be considered. If it matches the role of the agent which is associated with the Communicative Act, the `IContent` that the Application Adapter needs to process is retrieved using the `getRetriveContentAction()` method on the `CommunicativeAct` object. The same applies to Links in order to retrieve the associated actions for the link conditions. If the roles do not match, then the `IContent` objects that the Application Adapter needs to process are retrieved using the `getProcessCommand()` method of the `CommunicativeAct` object. The Application Adapter interface generator then automatically creates a mapping between these message

types and individual methods for handling them. This saves the developer a lot of typing because the big `switch` statement is now automatically generated. Only the actual implementation for the individual message types still needs to be provided manually. Also, since the mapping is now generated automatically, runtime errors due to typing errors in the string representations are avoided.

The generated Application Adapter also uses a string representation for the mapping between different types of `IContent` objects. But instead of using the value provided by the `toString()` method, it uses an auxiliary class (the `ActionNotificationStringFormatter`, which is located in the `org.ontoucp.application.server.interfaces` plug-in) to create a string representation for an action or notification. Both actions and notifications have a name and a list of named parameters and attributes which are specified in the Action-Notification Model. The string representation provided by that class is constructed from the name of the action or notification and also includes the names of its parameters and attributes, in contrast to the `toString()` representation that does not contain these values.

The Application Adapter interface stub generator also automatically creates a launch configuration for starting the UCP runtime. Some early versions of the "New Discourse Project" wizard (see Section 3.1) also generated a launch configuration for starting the UCP runtime, alongside the launch configuration for the GUI generation.[5] This approach had the disadvantage that the wizard would have to make assumptions regarding the names of the generated GUI and service projects, because those are created at some later point during GUI generation. The launch configuration for starting the runtime needs to reference these projects. If the GUI developer choses to change the names of the generated projects to something other than the default names, the launch configuration for starting the UCP runtime would also have to be changed accordingly. Also, the project generated by the "New Discourse Project" wizard is supposed to only contain the interaction model, which can be used as a base for multiple different generated GUIs (e.g., for devices with different screen sizes). The launch configuration for the UCP runtime is specific to a concrete GUI implementation and should, therefore, not be located in the discourse project. The Application Adapter interface stub generator places the generated UCP runtime launch configuration in the service project. Because the generation is triggered from the same launch configuration as the GUI generation, the Application Adapter interface stub generator is able to reference the correct GUI plug-ins (and also the service plug-in itself) in that generated launch configuration. This allows the UCP runtime to be started without the need to make any changes to the generated launch configuration.

3.2.3 Triggering the Generation

As mentioned above, version 1 of the UCP tools already contained a wizard for creating a service project. Because of this, it was first considered in this work to make the trigger for automatic generation of the Application Adapter interface stub a wizard as well. This option was tempting because some know-how about wizards in Eclipse was already gathered in the context of developing the "New Discourse Project" wizard.

However, the Application Adapter interface stub generator requires more input than the wizard for creating a new discourse project. While for that wizard a project name and Java package name are sufficient, the Application Adapter generator requires the following information:

[5] In most example UCP projects, the launch configuration for starting the UCP runtime is also located in the discourse project.

- The path of the Discourse Model.
- The role of the Application Adapter.[6]
- The project name and Java package name to be used for the generated code.
- The names of the generated GUI projects (those plug-ins need to be referenced in the launch configuration for starting the UCP runtime, which is also generated by the Application Adapter interface stub generator).

This is quite a lot of information that the GUI developer would have to provide. Also, because of the iterative nature of the UCP process, it is expected that the service project will be re-generated at some later point. If a wizard was used for the data input, the developer would have to provide the exact same values again. The Eclipse platform features some common dialogs for picking files of a certain type or choosing a project from the current workspace, but the resulting wizard would still not be very convenient to use.

In contrast to wizards, the property values of a launch configuration are persisted in a file. This makes it easier to re-run a task with the same settings as before. Therefore, it was considered to use a launch configuration for triggering the Application Adapter interface stub generation. This would have the advantage over a wizard that all input values would have to be specified only once. But this solution would still introduce some redundancy, as certain values (e.g., the names of the GUI projects) are already specified as part of the GUI generation launch configuration. If there were a separate launch configuration for the Application Adapter generator, the GUI developer would have to make sure that those values match in both launch configurations. Also, the launch configuration for the GUI generation already contains most of the information required for the Application Adapter generator. It was, therefore, decided to include the trigger for generating the Application Adapter interface stub as an additional generation step in the launch configuration for the GUI generation. This launch configuration is already established as a trigger for code generation for GUI developers. It also already provides access to the Discourse Model, which the Application Adapter generator requires. Only two properties (one for the name to be used for the generated service project, and another for the Java package name to be used for the generated code within that project) had to be added. The default values for those properties are provided by the "New Discourse Project" wizard. All generation steps are initially enabled for new discourse projects, as shown in Figure 3.5. Because of this, it is possible to generate both the GUI projects and the service project for a new discourse project without having to make any changes to the generated launch configuration.

The actual GUI generation is implemented in the `RenderingLaunchDelegate` in the plug-in `org.ontoucp.discourse.model2ui.rendering.ui`. When the GUI generation launch configuration is run by the GUI developer, the `launch()` method of that class is executed. This method in turn makes use of several auxiliary methods to perform the individual GUI generation steps. Several other classes are used for the actual code generation.

For creating the Application Adapter interface stub, a new method `generateApplicationStub()` was added to the `RenderingLaunchDelegate` class. This method is only invoked if the "Generate application stub" check box is checked in the GUI generation launch configuration. When executed, it creates a new instance of the `ApplicationAdapterInterfaceGenerator` class and invokes its `generate()` method, which generates or updates the service project.

[6] The Discourse Model must contain exactly two agents, one representing the "user" and one representing the "system". The discourse launch configuration allows the GUI developer to choose the role for which the GUI is generated. The other role is then represented by the Application Adapter.

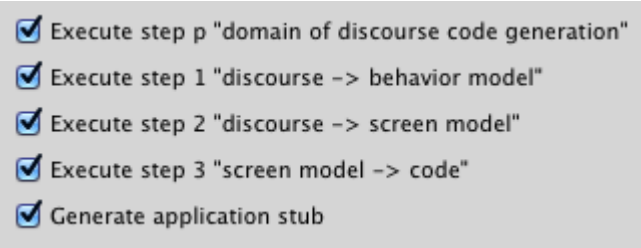

Figure 3.5: Selection of Generation Steps in the GUI Generation Launch Configuration.

3.2.4 Improving the Structure of the Generated Code

The first version of the Application Adapter interface stub generator put the whole generated code into one class that implements the Application Adapter interface. Also in most example UCP projects, the whole Application Adapter is implemented in a single Java class. This has several disadvantages:

- Putting the whole business logic, as well as all the variables that hold the state of a session, in one class results in a file with a very high number of code lines, which is hard to read and makes it hard for the developer to navigate through.
- The implementation of the `executeActionInternal()` method needs to perform different actions for different types of `IContent` messages, which are received as a method parameter. This is typically solved using a big `if/else` or `switch` statement, which again is hard to read because it has as many branches as there are Communicative Acts in the Discourse Model. Such a method will also most likely have a very high cyclomatic complexity.
- The class serves many different purposes: it is responsible for determining the type of the `IContent` message, actually handling each type of message, and holding the state of the user session. This violates the Single Responsibility Principle [Mar03].
- Updating the file when the service project is re-generated is not at all trivial because it contains both automatically generated code and user-generated code.

Since the objective for the development of the Application Adapter interface stub generator was to make it easier for the GUI developer to create an implementation of the Application Adapter, it was attempted to create a cleaner design for the generated Application Adapter code. In the second iteration, the generated code was split up into many smaller files that have just a single responsibility each:

- An "application" class, holding the state of the application that is shared between user sessions.
- A "session context" class, holding the state of the application that is related to one specific user session (e.g., some choices the user has made earlier in the interaction).
- Many individual "call handler" classes, each handling a specific type of `IContent` message.
- A "call handler factory" class that contains a mapping between different types of `IContent` messages and the associated call handler classes and constructs instances of call handlers.
- An "application adapter" class that implements the `ApplicationAdapterInterface`.

The application adapter class itself has become much more compact. When an instance is created, it retrieves the shared instance of the application class using the `getApplication()` method of its base class, `BasicApplicationAdapter`. It also creates a new instance of the session context class for the user session associated with this Application Adapter instance. The implementation of the `executeActionInternal()` method, provided by the Application Adapter interface stub generator as shown in Listing 3.4, consists of only a few lines of code. It uses an instance of the call handler factory class to get an instance of the right call handler class for the received type of `IContent` message. For obtaining the key for the map entry, the same class as in the Application Adapter interface stub generator (the `ActionNotificationStringFormatter`) is used to generate a string representation of the `IContent` message type. Finally, the `execute()` method of the returned call handler is invoked to perform some business logic associated with the message. If no call handler was found, a `NotImplementedException` is thrown. This warns the developer when, e.g., a Communicative Act was added to the Discourse Model but the Application Adapter has not been re-generated afterwards and no implementation for the new message type was provided.

```
@Override
public Object executeActionInternal(
      IContent call,
      Object content,
      ICommunicativeAct communicativeAct)
      throws NotImplementedException, ApplicationException {
   Object retVal = null;

   String callString = actionNotificationStringFormatter
      .getStringRepresentationForActionOrNotification(call);

   CallHandler handler = callHandlerFactory.getHandler(callString);
   if (handler != null) {
      retVal = handler.execute(content, communicativeAct);
   } else {
      throw new NotImplementedException(callString);
   }

   return retVal;
}
```

Listing 3.4: Implementation of the `executeActionInternal()` Method.

The call handler factory contains a static mapping between message types and call handlers. This mapping is provided by the Application Adapter interface stub generator and does not need to be updated manually by the developer. It is implemented using an actual `HashMap` instead of a big `if/else/switch` statement. When its `getHandler()` method is invoked, the call handler factory returns a new instance of the correct call handler class for the type of the `IContent` message being processed, based on that mapping. It also injects the shared instance of the application class and the session context instance into each created call handler instance, which provides the call handler implementations access to the application and session state these objects hold.

Listing 3.5 shows an example of a generated call handler class. The application and session context instances are injected by the call handler factory in the constructor and can then be accessed using getter methods provided by the base class `CallHandler`. The comment above

```
/**
 * Handles a "org.example.myDiscourse::startDiscourse" discourse call.
 */
class StartDiscourse extends CallHandler {
   /**
    * Initializes a new instance of the StartDiscourse class.
    */
   public StartDiscourse(
         Application application,
         SessionContext sessionContext) {
      super(application, sessionContext);
   }

   /**
    * Executes the discourse call.
    */
   public Object execute(
         Object content,
         ICommunicativeAct communicativeAct)
         throws ApplicationException, NotImplementedException {
      throw new NotImplementedException();
   }
}
```

Listing 3.5: Example of a Generated Call Handler Class.

the class declaration contains the original content specification string from the Discourse Model. This should make it easier for the developer to match a call handler class with the corresponding Communicative Act in the Discourse Model. The thrown `NotImplementedException` is supposed to be replaced with a real implementation by the developer.

Figure 3.6 shows the generated service project for a sample bike rental application (the Discourse Model of that project is shown in Figure 3.7). The folder `callHandlers` contains all individual call handler classes, named according to the content specification of the associated Communicative Acts in the Discourse Model. The types generated from the Domain-of-Discourse Model are placed in the `BikeRental` folder, which has the same name as that model file. The application adapter and call handler factory classes, as well as the base class for the call handlers, are located in the `infrastructure` folder. Files in that folder are not intended to be modified by the developer, as they will be overwritten each time the service project is re-generated. The application and session context classes are contained in the respective files. The files required for building and launching the Application Adapter (such as `build.properties` and `ExampleDiscourse UI.launch` are also provided by the Application Adapter interface stub generator.

The class name for each call handler class is derived from the string representation provided by the `ActionNotificationStringFormatter` for the corresponding message type. Because the class name needs to be a valid Java identifier, certain operators are replaced with a textual equivalent in order to make the resulting string more readable (e.g., "==" is replaced with "equals"). Then, all remaining characters that must not appear in Java identifiers are removed, and the remaining words are concatenated in camel case.[7]

[7] http://en.wikipedia.org/wiki/CamelCase

Figure 3.6: Example for a Generated Service Project.

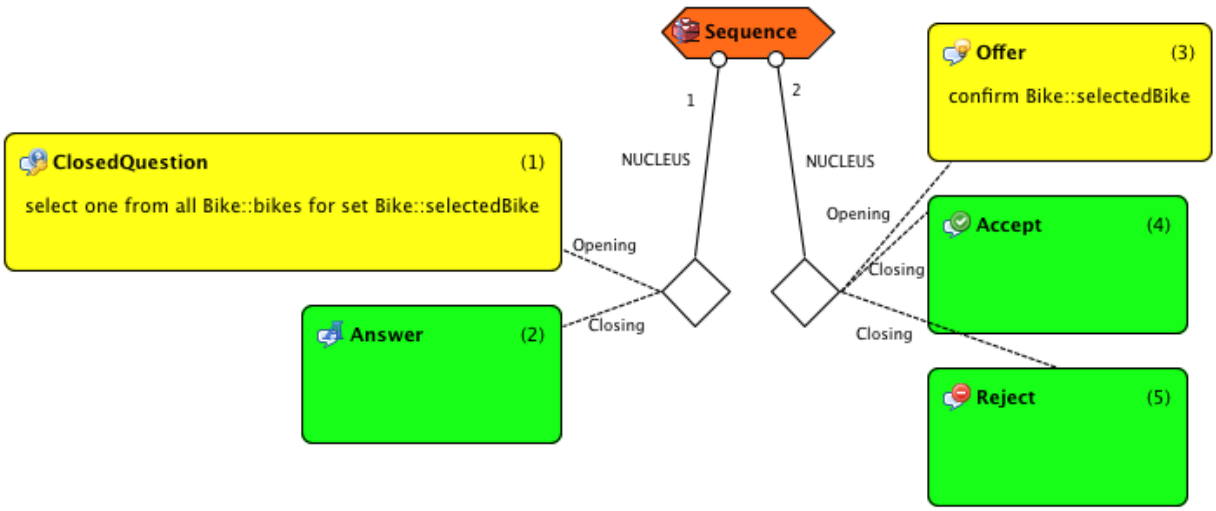

Figure 3.7: Screenshot of the Discourse Model.

In this example, the generated class `Confirm_ParameterSelectedBike_oneSelectedBike.java`, which is the topmost entry in the `callHandlers` folder in Figure 3.6, relates to the action `confirm` of the `Offer` Communicative Act (3) in Figure 3.7. The action itself is defined in the Action-Notification Model of the project and takes one parameter named `selectedBike` of type `Bike`. This type in turn is defined in the Domain-of-Discourse Model of the project.

The code generated by the Application Adapter interface stub generator has now a much cleaner structure. The readability of the code is increased because the originally large application adapter class is now split up in many smaller files. A GUI developer that needs to provide an actual implementation in the generated skeleton can now easier understand the purpose of each class. Also the design that makes use of multiple small call handler classes clearly indicates where the logic for handling a certain type of message is intended to be placed.

But most importantly, the generated service project can now be easily updated after a modification of the Discourse Model. All user-generated code is located in the application, session context and the individual call handler classes. The application adapter class, as well as the call handler factory, are not supposed to be modified by the developer. This means that they can safely be overwritten when the service project is re-generated. The generator checks whether the service project exists and only creates the application and session context classes if they have not been created yet. If they do already exist they are left untouched, preserving any custom implementation they might contain. Also, any existing call handler class is left unmodified. New call handlers (associated with `IContent` messages that have been added in the modification of the Discourse Model) are simply added to the project. The call handler factory (which contains the mapping) and the application adapter class are simply overwritten. Call handlers for messages which have been removed from the Discourse Model are kept in order to prevent accidental loss of code, but they no longer appear in the updated call mapping.

The Application Adapter interface stub generator now supports the iterative nature of the UCP process. The service project can be generated multiple times without losing any changes to the implementation made by the developer in the generated code skeleton.

4 Improvements to Existing Tools

This chapter lists the improvements made to certain existing tools in UCP. It outlines the design and implementation in version 1 of the UCP tools and provides details about the changes made in version 2.

4.1 Graphical Discourse Model Editor Improvements

The UCP tools comprise a graphical editor for the Discourse Model [FKP+09]. This editor is used in every project to create the initial version of the interaction model. As the GUI development process in UCP is iterative, it is also likely that this initial version of the model will be changed in further iterations, again using the aforementioned graphical editor. Good usability is, therefore, particularly important in this tool, because it is expected to be used frequently by the GUI developer. The following sections describe the improvements that have been made to the graphical Discourse Model editor in the course of this work.

4.1.1 Discourse Model Verification

One of the key features of the UCP tools is their ability to automatically generate a GUI from an interaction model. Because the input model for this generation process is provided by humans, it may contain errors. Such errors could then induce further errors in derived models and/or the final code of the GUI. Even worse, they might cause the generation process to get stuck. The earlier errors in the Discourse Model (that could cause the GUI generation to fail later on) are discovered, the more time is saved. In the best case they are detected while editing the model, before GUI generation has even started.

UCP provides a meta-model that specifies the elements of the interaction model and the relations between them [RSKF11]. Based on this specification, UCP provides a graphical model editor for the Discourse Model based on the Graphical Modeling Framework (GMF). This editor is inherently compliant with the meta-model, only allowing elements to be connected if a corresponding relation is specified in the meta-model.

However, there are some additional constraints that are not represented in the meta-model. For example, the meta-model defines a certain set of Communicative Acts (`OpenQuestion`, `ClosedQuestion`, `Answer`, `Request`, `Offer`, `Accept`, `Reject`, `Informing` and `Ok`). Adjacency

Pairs are used to relate one opening Communicative Act and zero to two closing Communicative Acts. But not all combinations make sense, e.g., if the opening Communicative Act is an `OpenQuestion` or a `ClosedQuestion`, then the closing Communicative Act has to be an `Answer`. Also, while a `Request` can have multiple closing Communicative Acts (e.g., `Accept` and `Reject`), it does not make sense to specify more than one `Answer` to, e.g., a `ClosedQuestion`. Table 4.1 lists the allowed combinations of Communicative Acts for each Adjacency Pair, based on [Sch10].

Opening Communicative Act Types	Closing Communicative Act Types
Open Question	Answer
Closed Question	Answer
Request	Accept Reject Accept & Reject
Offer	Accept Reject Accept & Reject
Informing	Ok

Table 4.1: Allowed Combinations of Communicative Acts.

Another example are the link types that are allowed as child nodes of a Discourse Relation. The meta-model defines a set of Discourse Relations, such as `Joint`, `Sequence` and `Condition`. A Discourse Relation can have some child elements which can be Adjacency Pairs or other Discourse Relations. For the connections to those children, certain types of links are specified in the meta-model, such as `Nucleus` or `Satellite`. Again, only some combinations (and cardinalities) are allowed. Table 4.2 shows the allowed combinations and also the allowed cardinalities.

Relation	Nucleus	Satellite	Tree	Then	Else
Joint OrderedJoint Switch Alternative Sequence	2...*	0	0	0	0
IfUntil	0	0	1	0...1	0...1
Condition	0	0	0	0...1	0...1
Background Elaboration Title	1	1	0	0	0
Result	1	0...1	0	0	0

Table 4.2: Allowed Cardinalities for Link Types.

The meta-model does not contain any representation of the restrictions described above.[1] While this might be considered a weakness of the meta-model, changing the meta-model would be quite a big undertaking that would be clearly out of scope for this work.

Both GMF and the underlying Eclipse Modeling Framework (EMF) feature model verification

[1] The classification of Communicative Acts by "assertion", "commissive" and "directive" (in contrast to "opening" and "closing") is, however, intentional according to [FPR+07].

through constraints.[2] Those constraints can be either "live constraints" (which are continuously evaluated while the model is being edited) or "batch constraints" (which are evaluated manually through a menu item in the editor). Constraints can be expressed in the form of Java code or OCL, have an associated severity (e.g., "error" or "warning") and can provide an error message to be displayed when the constraint is violated. They are well suitable to enforce certain rules in the Discourse Model because the evaluation can be triggered both manually by the user as well as programmatically, as a precondition for the automated GUI generation.

The graphical Discourse Model editor in version 1 of the UCP tools already contains many such constraints [Sch10]. The original idea was to add some new constraints in version 2 to cover additional aspects like those mentioned above. It was also considered to turn some batch constraints into live constraints in order to evaluate them earlier. This turned out to have a major drawback: the graphical GMF editor internally uses a `TransactionalEditingDomain`, in contrast to the `EditingDomain` used by the EMF editor. The key difference is that the graphical editor will instantly roll back every change to the model that violates any constraint with the severity "error". This is potentially a big usability issue because it is impossible to make certain changes to the Discourse Model without having any transitional states that do not pass verification. For example, just adding a new `Joint` Discourse Relation element would already violate a constraint, because such elements are required to have at least two child nodes connected via `Nucleus` links. Those links can only be added after the element was created, but the change in which the element is added is immediately rolled back by the editor because the constraint regarding its child nodes is not fulfilled. It would be possible to use the EMF editor (which provides a tree view of the Discourse Model) to make such changes, but then it would be pointless to have a graphical editor, which is supposed to be more convenient, but is actually not usable for some basic editing tasks. Reducing the severity of the constraints to "warning" or below would also keep the graphical editor from rolling back. But it makes sense to report such issues (that would make GUI generation fail) as errors.

Therefore, another solution was chosen: rather than adding constraints that verify the model later on, the graphical editor was modified to offer fewer invalid options to the user. Menus for creating links to child elements have been changed to display only those elements that are allowed to appear at that position in the Discourse Model. For example, when creating a link from a `Joint` Discourse Relation element, the editor now offers `Nucleus` links as the only allowed option for the link type, as shown in Figure 4.1. This makes the GUI of the graphical Discourse Model editor simpler because it now has fewer options in various menus that the GUI developer needs to understand. The developer can make fewer mistakes when editing the Discourse Model because elements that are not allowed at a certain position are not shown at all. In many cases, this leaves him with a single option, as in the example with the `Joint` element.

Which types of child links can be created for which type of Discourse Relation in the graphical editor can be restricted by specifying creation constraints for links. This can be done in the `links` section of the `discourse.gmfmap` file for any of the link types. Listing 4.1, e.g., shows the snippet for the `Tree` link, which needs to have an `IfUntil` as its parent.[3]

The source end constraint ensures that the parent of any `Tree` link is an `IfUntil` element. The target end constraint prevents the link from pointing back to its parent, which would form a loop.

[2] Although this is actually a verification of the model, the term "validate" is used throughout Eclipse and is well established in the Eclipse community.

[3] The expression "<>" represents the inequality operator in OCL ("<>"), but the less-than and greater-than characters must be encoded since the `discourse.gmfmap` file is an XML file.

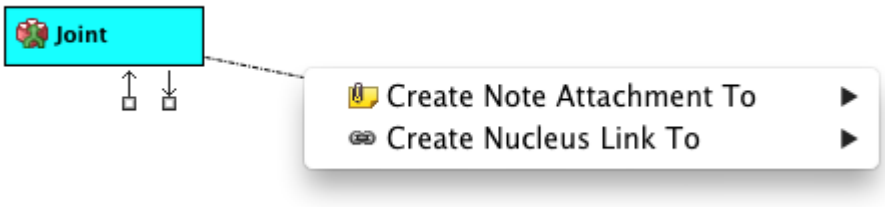

Figure 4.1: Link Types for Child Nodes of `Joint` Discourse Relations.

```
<links>
   <domainSpecialization body="self.type = LinkType::TREE"/>
   ...
   <creationConstraints>
      <sourceEnd body="self.oclIsTypeOf(IfUntil)"/>
      <targetEnd body="self &lt;&gt; oppositeEnd"/>
   </creationConstraints>
</links>
```

Listing 4.1: Example of a Link Creation Constraint.

The constraints are written in the Object Constraint Language (OCL) and can be any valid OCL expression. This allows for writing more complex constraints that also limit the cardinality of child links being created. The example in Listing 4.2 limits the maximum number of `Tree` links of an `IfUntil` to `1`:

```
self.oclIsTypeOf(IfUntil)
   and self.children->select(type = LinkType::TREE)->isEmpty()
```

Listing 4.2: Cardinalities in Link Creation Constraints.

Such link creation constraints are evaluated by the GMF editor prior to displaying the menu with possible options for link types to the GUI developer. If they evaluate to `false`, the respective entry is hidden from the menu, making it impossible to create a link of such a type with this editor.

The allowed types of Communicative Acts can be restricted for the link connections of opening and closing Communicative Acts in a similar fashion. This again prevents invalid combinations from being created in the first place, instead of having to repair them later. The following link constraint was added for the target end of the link to opening Communicative Acts:

```
self.oclIsTypeOf(OpenQuestion)
   or self.oclIsTypeOf(ClosedQuestion)
   or self.oclIsTypeOf(Request)
   or self.oclIsTypeOf(Offer)
   or self.oclIsTypeOf(Informing)
```

Listing 4.3: Link Creation Constraint for Opening Communicative Acts.

A similar constraint was created for the closing Communicative Acts:

```
self.oclIsTypeOf(Answer)
   or self.oclIsTypeOf(Accept)
   or self.oclIsTypeOf(Reject)
   or self.oclIsTypeOf(Ok)
   or self.oclIsTypeOf(Informing)
```

Listing 4.4: Link Creation Constraint for Closing Communicative Acts.

After having made changes to the graphical editor in `org.ontoucp.discourse.model.diagram`, it is important to re-generate the actual editor code (`org.ontoucp.discourse.model.edit` and `org.ontoucp.discourse.model.editor`) to reflect these changes. In order to do so, first the generator model (`discourse.gmfgen`) needs to be updated by right-clicking the `discourse.gmfmap` file and choosing `Create generator model...` Afterwards, the editor code can be re-generated by right-clicking the generator model file and choosing `Generate diagram code`.

The method described above allows for specifying which links can be created from any source element, but it is not suitable for imposing restrictions upon the allowed target element types. Another method must be used to prevent, e.g., an `Answer` to be created for an opening Communicative Act. In the first attempt, the `getMATypesForTarget()` method in the generated `AdjacencyPairEditPart` class of the graphical Discourse Model editor was modified. The method returns a list of `DiscourseElementTypes` that can be used as target for either opening or closing Communicative Act links. The snippet in Listing 4.5 shows the original list for the opening Communicative Act where the elements that are only allowed for closing Communicative Acts have been commented out:

```
types.add(DiscourseElementTypes.Request_2013);
types.add(DiscourseElementTypes.ClosedQuestion_2014);
types.add(DiscourseElementTypes.OpenQuestion_2015);
types.add(DiscourseElementTypes.Offer_2016);
types.add(DiscourseElementTypes.Informing_2017);
// types.add(DiscourseElementTypes.Accept_2018);
// types.add(DiscourseElementTypes.Answer_2019);
// types.add(DiscourseElementTypes.Reject_2020);
// types.add(DiscourseElementTypes.Ok_2021);
```

Listing 4.5: Specification of the Allowed Opening Communicative Act Types.

It is important to annotate the changed method with `@generated not` in order to prevent it from being reverted to its default implementation when the Discourse Model editor is re-generated using its generator model in `org.ontoucp.discourse.model.diagram`.

With this approach, only the relevant types of Communicative Acts appear in the menu when a new opening/closing link from an Adjacency Pair is created. But this solution still does not take already created Communicative Acts of an Adjacency Pair into account. Considering the allowed combinations listed in Table 4.1, the user should be prevented from creating combinations with closing Communicative Acts which are not allowed. For example, if an Adjacency Pair with an opening Communicative Act of type `Request` already has one closing Communicative Act of type `Accept`, it should be impossible to add another closing Communicative Act of type `Accept` (but adding a `Reject` should still be allowed).

Because the list assembled by the `getMATypesForTarget()` method cannot be related to a specific instance of an Adjacency Pair, another approach was taken by modifying the method `getTypesForTarget()` of the `DiscourseModelingAssistantProvider` class. This method receives the source element of the link (which is an Adjacency Pair) as a parameter and can then check which Communicative Acts are already connected to that Adjacency Pair. It then creates a list of allowed element types for the link target, similar to the `getMATypesForTarget()` method. The implementation was extended to take the constraints from Table 4.1 into account. Element types that are not allowed are simply not included in the returned list. Listing 4.6 shows a part of the implementation that makes sure that only `Accept`, `Reject`, `Ok`, `Informing` or both `Accept` and `Reject` are included in the list of allowed elements if the opening Communicative Act is a `Request`.

```
...
if (openingCommunicativeAct instanceof Request) {
   // A request can have an accept, reject, ok, informing, or both
   // accept and reject as closing Communicative Acts.
   EList<CommunicativeAct> closingCommunicativeActs = adjacencyPair.
      getClosingCommunicativeActs();
   if (closingCommunicativeActs != null) {
      if (closingCommunicativeActs.isEmpty()) {
          types.add(DiscourseElementTypes.Informing_2017);
          types.add(DiscourseElementTypes.Accept_2018);
          types.add(DiscourseElementTypes.Reject_2020);
          types.add(DiscourseElementTypes.Ok_2021);
      } else if (closingCommunicativeActs.size() == 1) {
          if (closingCommunicativeActs.get(0) instanceof Accept) {
              types.add(DiscourseElementTypes.Reject_2020);
          } else if (closingCommunicativeActs.get(0) instanceof Reject)
             {
              types.add(DiscourseElementTypes.Accept_2018);
          }
      }
   }
}
...
```

Listing 4.6: Filtering the Allowed Types of Communicative Acts.

Because the `getTypesForTarget()` implementation allows for specifying a more fine-grained filter, the implementation of the `getMATypesForTarget()` method in the `AdjacencyPairEditPart` class was then changed to return an empty list, as shown in Listing 4.7. The GMF editor will only show the allowed types of Communicative Acts to the GUI developer that are contained in the list as provided by the `DiscourseModelingAssistantProvider`.

4.1.2 Toggling the Associated Agent of Discourse Relations

A Discourse Model must specify exactly two agents, one representing the user and the other the system. Every Communicative Act is associated with either one of these two agents. The associated agent is the party that performs that Communicative Act (e.g., asks or answers a question). Which one of the two agents is associated with a Communicative Act is specified

```
/**
 * @generated not
 */
public List<IElementType> getMATypesForTarget(IElementType
    relationshipType) {
    // The DiscourseModelingAssistantProvider will
    // fill in all available target types.
    return null;
}
```

Listing 4.7: Final Implementation of the `getMATypesForTarget()` Method.

in a property of the model element representing the Communicative Act. When the element is selected in the editor, the value of this property can be modified using the `Properties` panel of the Eclipse IDE.

The graphical Discourse Model editor features a nice addition that makes it easier to change the associated agent of a Communicative Act. When the element representing the Communicative Act is double-clicked, its associated agent is toggled to become the other one. If no agent has been specified yet for that element, the first agent specified in the Discourse Model (which is the one representing the user by default) is assigned. This can then easily be changed to the other agent by double-clicking on the Communicative Act element another time.

This specific behavior is implemented as a custom edit policy, defined in the `CAOpenEditPolicy` class of the `org.ontoucp.discourse.model.diagram` plug-in [Gro09, Chapter 9.2.3]. When executed, it basically retrieves the model element (the Communicative Act) associated with the editor command, gets the two agent elements specified in the Discourse Model and checks which agent matches the value of the `belongsTo` property of that Communicative Act. It then creates and returns a `SetValueCommand` that changes the value of that property to become the other agent. The command is than evaluated by EMF, which effectively changes the associated agent of the Communicative Act.

In order to work, this custom edit policy is installed in `createEditPolicies()` method of the `DiscourseEditPolicyProvider` class, as shown in Listing 4.8. The `OPEN_ROLE` value of the `EditPolicyRoles` enumeration is the one associated with the double-click action in the graphical editor.

The `DiscourseEditPolicyProvider` itself is plugged into the graphical discourse model editor by specifying the "edit policy provider" extension as shown in Listing 4.9 in the `plugin.xml` of the plug-in `org.ontoucp.discourse.model.diagram` [Gro09, Chapter 10.9.3]:

But Communicative Acts are not the only elements of a Discourse Model that have an associated agent. The Discourse Relations `Switch`, `Alternative`, `IfUntil`, `Condition`, `Elaboration` and `Result` can specify a condition that also needs to be evaluated by one of the interacting agents. It would be favorable if the agent associated with an instance of one of these Discourse Relations could be toggled in the same fashion as for a Communicative Act. This feature was however missing in the graphical Discourse Model editor in version 1 of the UCP tools.

This behavior was added to the UCP tools by creating another edit policy for toggling the associated agent of a Discourse Relation. The new `DiscourseRelationOpenEditPolicy` class

```
@Override
public void createEditPolicies(EditPart editPart) {
   // Check if our model element is a view
   // (otherwise don't return a command).
   final Object model = editPart.getModel();
   if (model instanceof View) {
      final EObject obj = ((View) model).getElement();

      // Install edit policy for Communicative Acts.
      if (obj instanceof CommunicativeAct) {
         editPart.installEditPolicy(
            EditPolicyRoles.OPEN_ROLE,
            new CAOpenEditPolicy());
      }
   }
}
```

Listing 4.8: Installation of a Custom Edit Policy for Communicative Act Elements.

```
<extension
   point="org.eclipse.gmf.runtime.diagram.ui.editpolicyProviders">
   <editpolicyProvider
         class="org.ontoucp.discourse.model.discourse.diagram.custom.
            DiscourseEditPolicyProvider">
      <Priority name="Lowest"></Priority>
   </editpolicyProvider>
</extension>
```

Listing 4.9: DiscourseEditPolicyProvider Specification.

uses similar code as the `CAOpenEditPolicy` class to change the value of the `agent` property of the Discourse Relation. It also checks whether the type of the Discourse Relation is one of those that can have an associated agent (`Joint`, `OrderedJoint`, `Title`, `Sequence` and `Background` Discourse Relations do not have an associated agent). Only for those types a `SetValueCommand` is created. This new edit policy is added to the registration in the `DiscourseEditPolicyProvider` class, as shown in Listing 4.10:

In version 2 of the UCP tools, the agent associated with a Discourse Relation can now be toggled as easily as for Communicative Acts.

4.1.3 Prevent Deletion of Content Compartments

In the visual representation of a Communicative Act in the graphical Discourse Model editor, its content is displayed (and can be edited) in a GMF compartment [Gro09, Chapter 11.1.3]. This compartment can be selected independently from its parent element in the graphical editor, as shown in Figure 4.2. If the user then presses the delete key (or chooses `Delete from Model` from the context menu), the default implementation of the GMF editor will remove the content compartment from its parent element. Unfortunately, there is no way to get the deleted content

```
@Override
public void createEditPolicies(EditPart editPart) {
   // Check if our model element is a view
   // (otherwise don't return a command).
   final Object model = editPart.getModel();
   if (model instanceof View) {
      final EObject obj = ((View) model).getElement();

      // Install edit policy for Communicative Acts.
      if (obj instanceof CommunicativeAct) {
         editPart.installEditPolicy(
            EditPolicyRoles.OPEN_ROLE,
            new CAOpenEditPolicy());
      }

      // Install edit policy for Discourse Relations.
      if (obj instanceof DiscourseRelation) {
         editPart.installEditPolicy(
            EditPolicyRoles.OPEN_ROLE,
            new DiscourseRelationOpenEditPolicy());
      }
   }
}
```

Listing 4.10: Installation of a Custom Edit Policy for Discourse Relation Elements.

compartment back in the graphical editor, other than removing the entire element, creating a new one and re-creating all links the original element had to other elements.

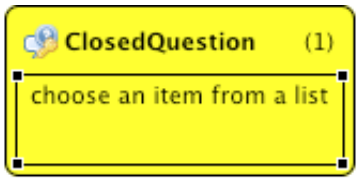

Figure 4.2: Selected Content Compartment of a Communicative Act.

Unintended deletion of a content compartment can be prevented by changing the implementation of the semantic editing policy class for its parent's element type. Listing 4.11 shows the overridden `getSemanticCommand()` method that doesn't return any command for a `DestroyRequest`. This method has to be inserted in every content compartment item semantic edit policy class, e.g., `ClosedQuestionContentCompartmentItemSemanticEditPolicy` for the `OpenQuestion` Discourse Relation:

The overridden method is annotated with `@generated not` in order to prevent the change from being reverted when code for the Discourse Model editor is re-generated.

Note that this change only prevents the content compartment from being deleted independently of its parent element. The element representing the Communicative Act itself remains removable.

```
/**
* @generated NOT
*/
@Override
protected Command getSemanticCommand(IEditCommandRequest request) {
   return request instanceof DestroyRequest
      ? null
      : super.getSemanticCommand(request);
}
```

Listing 4.11: Custom `getSemanticCommand()` Implementation.

4.1.4 Discourse Editor Auto-Arrangement

Graphical model editors based on GMF support auto-arrangement of elements. By a single click on the `Arrange All` menu entry, the editor will move the nodes so that they would no longer overlap and the links between nodes would cross each other as little as possible. The visual representation of the Discourse Model in its graphical editor is a tree-shaped collection of nodes which are connected by links. The auto-alignment feature of GMF by default oriented this tree in a way such that its root node was placed on the bottom of the screen. This is not very intuitive because this way the Communicative Acts which appear later in the course of an interaction are placed on top.

This behavior was modified by using a top-down layout provider, which is provided by GMF [Gro09, Chapter 10.4.10]. This is done by specifying a layout provider extension in the `plugin.xml` of the discourse diagram editor plug-in (`org.ontoucp.discourse.model.diagram`), as shown in Listing 4.12:

```
<extension point="org.eclipse.gmf.runtime.diagram.ui.layoutProviders">
   <?gmfgen generated="false"?>
   <layoutProvider
         class="org.eclipse.gmf.runtime.diagram.ui.providers.
            TopDownProvider">
      <Priority name="High"></Priority>
   </layoutProvider>
</extension>
```

Listing 4.12: Top-Down Layout Provider Extension.

This makes the diagram nodes arranged top-down when the `Arrange All` feature is used. The statement `<?gmfgen generated="false"?>` makes sure that the custom layout provider specification is not discarded when the discourse diagram editor plug-in is re-generated using its generator model.

4.2 Standalone Eclipse RCP Application

The UCP tools consist of a set of plug-ins for the Eclipse platform. Those plug-ins can be added to existing Eclipse installations, which then gain the capabilities of the UCP tools. There are

several ways how the UCP tools can be installed. The easiest option is to add the URL of the UCP update site to Eclipse.[4] The UCP plug-ins will then appear in the `Install New Software...` dialog of Eclipse and can be installed from there. This is the common way for extending Eclipse with additional plug-ins, but requires some manual steps to be performed by the user. Also, the user needs to know the URL of the UCP update site. As an alternative, the plug-ins could also be manually downloaded and copied to the `plugins` directory of the Eclipse installation.

The distribution of the UCP tools can be made even more convenient for the GUI developer by creating an Eclipse Rich Client Platform (RCP) application. An Eclipse RCP application consists of a set of Eclipse plug-ins or features that offer a certain functionality based on the Eclipse platform. Such an application is defined in a so-called "product definition", which specifies the plug-ins and features to be included in the application. The main view (the application window) of the application is defined by a "workbench". Eclipse allows for extending the default Eclipse workbench in RCP applications (which will give them the same look and feel as the Eclipse IDE, extended with some custom features). Alternatively, a custom workbench can be implemented in an Eclipse plug-in, which allows for even greater customization. Eclipse can then export the application easily for multiple supported platforms (e.g., Windows, Linux or OS X).[5] This creates a ZIP file that contains a runnable Eclipse distribution that contains all the plug-ins specified in the product definition. The ZIP file can then be easily distributed and requires no other installation steps other than extracting the archive. This solution even allows updates to be deployed conveniently and without user interaction—Eclipse will automatically fetch them from the update sites of the packaged plug-ins once they are available.

The UCP tools already contained a product definition, but the export was broken since after one past upgrade of the Eclipse platform. This was repaired by updating the list of Eclipse features in the product definition. Also, all required plug-ins need to be listed in the `MANIFEST.MF` of the `org.ontoucp.workbench` plug-in in order for the export to work. UCP can now again be exported as standalone Eclipse RCP application (this was successfully tested on Windows, Linux and OS X).

The UCP application also features a custom Eclipse workbench. A workbench defines the overall layout of the application window, i.e., the available perspectives and views, the panels which are initially displayed as well as their location, and the menus and toolbars. Eclipse is a very feature-rich and multi-purpose IDE, but most of its features are not required in the context of developing a GUI using the UCP tools. The custom UCP workbench provides a stripped-down user experience that by default hides most of the panels that are not of interest for UCP development. Instead, it promotes those which are related to editing Discourse Models (such as the `Properties` panel and the custom `CA Content Inspection` panel for viewing the content specification of Communicative Acts). This makes the application window less cluttered and allows the developer to focus more easily on the GUI development with UCP. Also, creating a new discourse project is made more user-friendly with an extra toolbar button that directly opens the "New Discourse Project" wizard. Figure 4.3 shows the default view of the UCP workbench.

The UCP application also copies the bundled default Device Specifications to the workspace when it is started for the first time. This eliminates the need to copy them manually to the workspace or change some paths in the launch configuration of a discourse project. The inclusion of the Device Specifications makes the UCP application self-contained, no external resources are required in order to get a project running. The location of the Device Specification matches the

[4] http://ucp.ict.tuwien.ac.at/eclipse-update/

[5] This has to be done for each target platform individually.

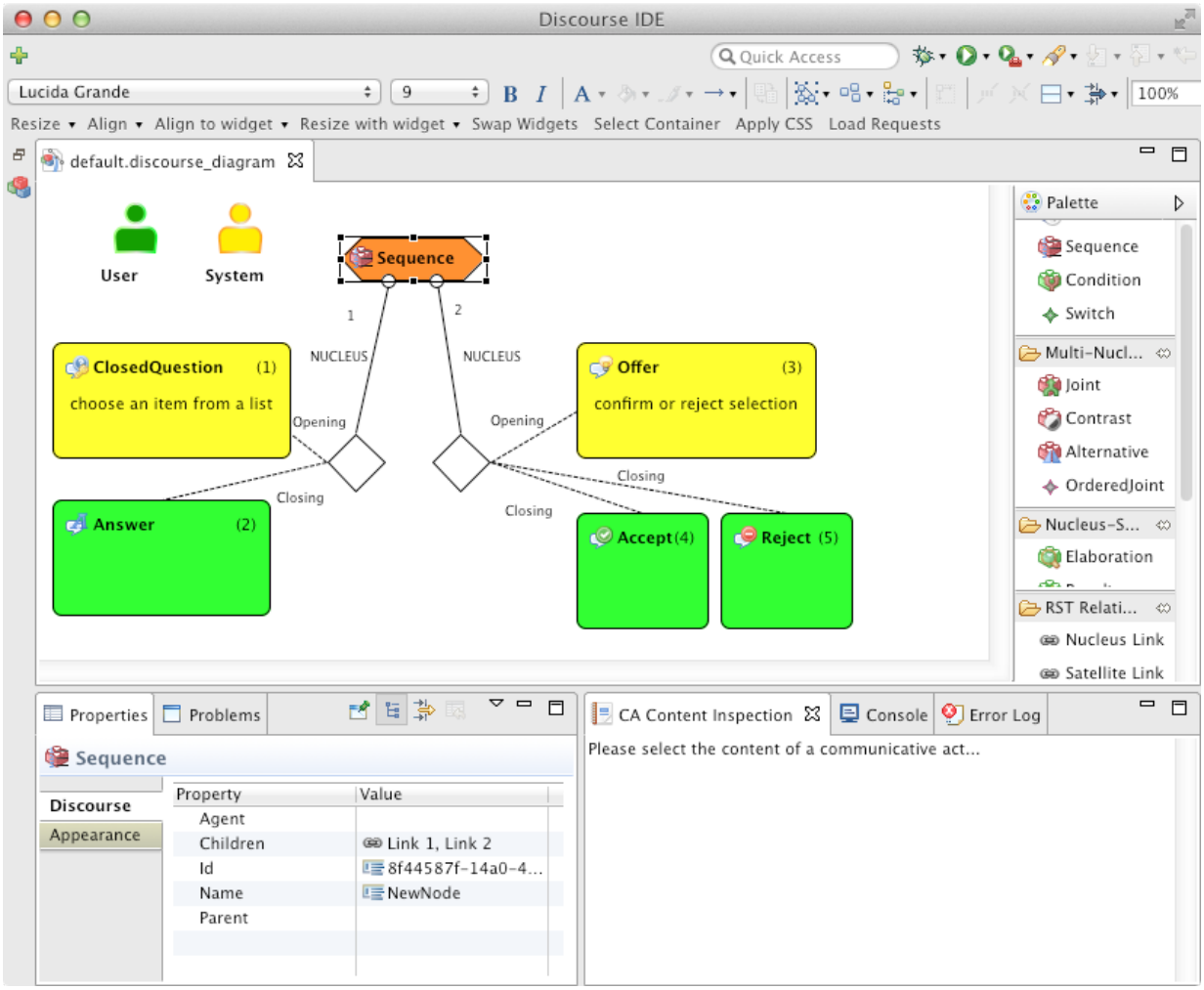

Figure 4.3: The Custom UCP Workbench.

default value generated by the "New Discourse Project" wizard introduced in Chapter 3, so no changes to launch configuration properties need to be done when a project is created using the wizard.

4.3 GUI Generation Launch Configuration

In UCP, the automatic GUI generation is triggered via a custom Eclipse launch configuration. Such a GUI generation launch configuration has a certain set of custom properties and the UCP tools provide a GUI for modifying them (see Figure 2.3). The launch configuration can be executed by the user in the `Run` menu of Eclipse. This triggers the execution of a so-called launch delegate, which then performs the automatic GUI generation. The generation of the GUI is implemented in the `RenderingLaunchDelegate` class of the `org.ontoucp.discourse.model2ui.rendering.ui` plug-in, which also provides the custom editor.

The custom launch configuration for the GUI generation in UCP includes many different properties which specify the location of the input models and the project names, namespaces and

Java source folder names for the generated output. To save some time, the launch configuration editor allows the GUI developer to choose which steps of the GUI generation process shall be performed. For example, if no changes have been made to the Domain-of-Discourse Model, then it is not necessary to re-generate the resulting Java classes which represent the domain objects specified in that model.

While the implementation of the launch configuration for GUI generation in version 1 of the UCP tools was working well, some input fields in the launch configuration editor have been found to be potentially confusing for the GUI developer. For example, the names of the output projects were not specified explicitly in a separate input field, but implicitly, as part of the path to some intermediate model files, as shown in Figure 4.4. This was not only hard to understand for novice users, but was also error-prone because the exact same project name had to be used in multiple properties of the launch configuration. If the values were different, GUI generation would fail.

Figure 4.4: Implicit Output Project Name Configuration in Version 1 of the UCP Tools.

This was cleaned up in the course of this work by introducing separate input fields for the output project names for each generated project, on the respective tabs on the launch configuration dialog. The source paths are now relative to these project names, which removes the possibility to enter non-matching project names in those fields. Now it is much easier to understand what a particular field in the launch configuration dialog means and where the names of the generated output come from. Figure 4.5 illustrates the output project configuration in version 2 of the UCP tools.

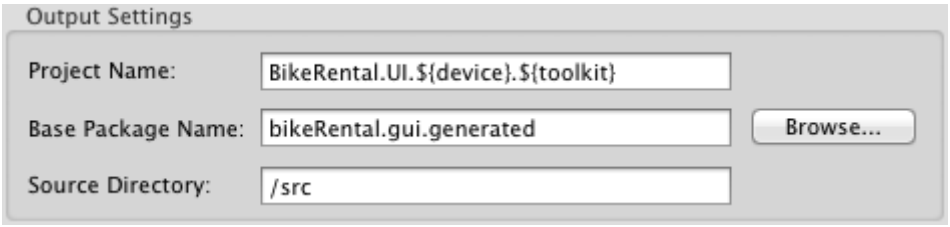

Figure 4.5: Explicit Output Project Name Configuration in Version 2 of the UCP Tools.

The properties for the output project name already supported the placeholder `${toolkit}` in order to allow multiple GUIs (based on the same Discourse Model) to be created for different supported GUI toolkits.[6] The launch configuration editor also allows for choosing a Device Specification, which allows different GUIs to be generated for devices with different characteristics (such as screen size). Many example projects actually contain multiple copies of the GUI generation launch configuration for every device type, in order to be able to generate a GUI for that device without having to edit the output project names in the launch configuration. This was made easier by introducing a second placeholder, `${device}`, for the output project name. It is automatically replaced by the name of the device which is selected in the launch configuration

[6] At the time of writing, only the `HTML` toolkit is supported.

when the GUI is generated. This makes the multiple (and except for the different device and output project name settings identical) launch configurations obsolete.

Additionally, in version 2 of the UCP tools the explicit settings for the Domain-of-Discourse code generation have been removed from the launch configuration (and also from the launch configuration editor). The Domain-of-Discourse Models are linked to the Discourse Model anyway, and the Discourse Model is available in the context of the automatic GUI generation, which avoids the need for explicit settings. This also removes a potential cause of errors during GUI generation, related to inconsistent settings for the Domain-of-Discourse code generation.

4.4 Copying Transformation Rules

In the Rendering Model, the transformation rules all have unique IDs. When such a rule is copy&pasted in the model editor for the Rendering Model, the default implementation provided by EMF copies all its attributes, including the ID. This, of course, makes the model invalid until the ID of the copied rule is manually changed, because two rules with the same ID are not allowed.

A custom implementation of the copy command was provided in the `RuleItemProvider.java` class in the `org.ontoucp.discourse.model2ui.rendering.model.edit` plug-in, in order to simplify copying of transformation rules. The custom `InitializeCopyCommand`, shown in Listing 4.13, uses the base implementation to copy all attributes. Then it creates a new UUID and assigns it to the copied transformation rule as its unique identifier. This allows transformation rules to be copied in the Rendering Model editor without having to manually specify a unique ID for the copied rule.

```
/**
 * @generated NOT
 */
@Override
protected Command createInitializeCopyCommand(
      EditingDomain domain,
      EObject owner,
      Helper helper) {
   return new InitializeCopyCommand(domain, owner, helper) {
      @Override
      protected void copyAttributes() {
         // Copy all attributes.
         super.copyAttributes();

         // Set a custom ID.
         EAttribute idAttribute = copy.eClass().getEIDAttribute();
         String value = UUID.randomUUID().toString();
         copy.eSet(idAttribute, value);
      };
   };
}
```

Listing 4.13: Custom `InitializeCopyCommand` Implementation.

The custom implementation is annotated with `@generated not` in order to prevent it from being reverted when the Rendering Model editor is re-generated using its generator model.

5 Evaluation

This chapter demonstrates how the improved tool support in UCP can be used to realize a discourse project. The following section provides a walkthrough of a sample project, this time using version 2 of the UCP tools. Section 5.2 then provides a comparison between version 1 and version 2 of the UCP tools.

5.1 Sample Project Walkthrough

Section 2.1.4 provides a list of steps that the GUI developer needs to perform using version 1 of the UCP tools in order to create a runnable GUI prototype for a new discourse project. The tasks in that list involve the creation of an interaction model (with all related types of files), and automatic generation of the GUI code based on that model. An Application Adapter for the project was created manually. After completing each step, the UCP runtime could be started and the generated GUI of the discourse project could be opened in a Web browser window.

The same process was repeated, this time using version 2 of the UCP tools. As with version 1, a new discourse project was started from scratch. The following list presents the steps that are required in order to create a runnable GUI for a new project in version 2 of the UCP tools:

1. Create a new project using the "New Discourse Project" wizard.
2. Create some domain objects in the Domain-of-Discourse Model.
3. Create some Communicative Acts in the Discourse Model using the graphical editor.
4. Perform code generation via the launch configuration, using all default values as given.
5. Provide an implementation for the individual generated call handler classes in the Application Adapter.
6. Start the UCP runtime for the generated GUI using the launch configuration in the generated service project, again using all default values as given.

After these steps have been performed, the GUI of the discourse project can be accessed using any Web browser on desktop as well as mobile devices. This has been confirmed to work with

the Windows and OS X versions of the standalone RCP application by the author and two other GUI developers that already had some experience with version 1 of the UCP tools.

Apart from being able to generate the GUI and Application Adapter interface stub for new discourse projects, it was verified that version 2 could still successfully generate the GUI for a few example projects which have been initially created using version 1 of the UCP tools. This required some slight adaptations to the launch configurations, but not to the interaction models.

5.2 Comparison

Looking back to the original list in Section 2.1.4 (which had 18 items), the same task can be achieved in much fewer steps (and also faster) using version 2 of the UCP tools. The following list sums up the major improvements in comparison to version 1:

- Steps 1 to 9, as well as Step 13 (related to setting up a new, empty discourse project), have become obsolete. The "New Discourse Project" wizard can be used instead to achieve the same result in a single step.
- Step 12 (copying the Device Specifications to the Eclipse workspace) is no longer necessary in the context of the standalone RCP application, which bundles the default Device Specifications and automatically places them in the workspace.
- Steps 15 and 16 (creating an implementation of the Application Adapter interface) have become much easier due to the new Application Adapter interface stub generator. The generated code already contains the logic for switching between the actions and notifications specified in the Discourse Model, the developer needs to provide only the actual application logic. The generator also automatically creates a launch configuration for the UCP runtime, which makes Step 17 obsolete.

The remaining items are mostly related to editing the interaction model and implementing the business logic of the Application Adapter. These tasks involve a lot of creative work and are very specific to the application being developed, so they can hardly be automated by tools. Still, the improvements described in Chapter 4 make these tasks easier to do and also easier to focus on.

All things considered, version 2 of the UCP tools provides a better user experience when compared with version 1. It saves the GUI developer multiple manual steps and requires less input that could be prone to errors, particularly in the context of setting up the launch configurations. Also, the usability of the graphical model editors has been improved. Especially for GUI developers with little or no experience with UCP, version 2 of the UCP tools makes it less likely to run into a confusing situation where the developer could do something wrong in the technicalities.

6 Discussion and Outlook

The main objective of this work was to improve the tool support in UCP in order to cover as much as possible of the GUI development process, aiming for better usability and a less steep learning curve for novice users. Version 1 of the UCP tools already covered many important aspects of the process, but some things were still unnecessarily hard to achieve. The author has made his own experiences in the course of a seminar as a member of a team attempting to develop a small application using UCP. Just to get to a first version of our Discourse Model without any verification errors required the team to consult their tutors many times. We had encountered many small problems (e.g., forgetting to add an Action-Notification Model, forgetting to link the Action-Notification Model to the Discourse Model—the list goes on) that were finally all easily resolved. However, the same problems may not be easily identified by someone not familiar with the UCP tools.

At the very beginning of this work, a list of improvements to be done to the UCP tools was assembled. While it was not difficult to identify what the bigger "missing pieces" of the UCP tools were (better support for creating a new project, triggering GUI generation and creating an Application Adapter), it was interesting to observe how the options for achieving these objectives evolved during their development. The first version of the "New Discourse Project" wizard just created a new project and the required model files (the Discourse Model, Domain-of-Discourse Model and Action-Notification Model). Later on it was discovered that it would be even more convenient to include the launch configuration for generating the GUI and the default Rendering Rules file. The Rendering Rules file is not even mandatory for each discourse project, but providing all kinds of files related to a discourse project would probably give inexperienced users a better overview over all available customization options. Otherwise they would need much more in-depth knowledge about UCP to understand that there is a possibility to add a certain type of file to the project in order to be able to do some further customizations.

An early version of the "New Discourse Project" wizard included the functionality of the "New Service Project" wizard from version 1 of the UCP tools. This was clearly an improvement, requiring the GUI developer to run only one wizard instead of two in order to create both projects. It was recognized later that it is actually beneficial to move the generation of the service project from the "New Discourse Project" wizard back to the GUI generation launch configuration. This way the generator for the service project gains access to the Discourse Model and can then generate a much better Application Adapter interface stub that is related to that Discourse Model, instead of just an empty, generic skeleton.

The changes regarding the GUI generation launch configuration are not completely backward compatible with existing launch configurations that have been created using version 1 of the

UCP tools. Most properties have remained the same in version 2, but some which are related to the names of the output artifacts (such as the project and Java package names for the generated projects) have been renamed and some new properties have been introduced. When a launch configuration created with version 1 is opened in version 2 of the UCP tools, the values of these properties need to be updated manually. This is inconvenient, but having properties with a clear meaning and purpose was considered to be more important, since such a trade-off makes the GUI generation launch configuration editor more self-explanatory (which is helpful especially for new GUI developers). Furthermore, the undesired manual adaptation only needs to be done once.

There is still enough room for further improving the usability of the UCP tools. For example, the custom workspace of the standalone RCP application could be streamlined even further. The default toolbar still contains many items that are only related to specific model editors. Those could be made initially hidden and only displayed while the associated editor is open. Also, while the initial perspective is suitable for editing the interaction models, it is less convenient when writing Java code for the Application Adapter implementation. For this use case, it would be favorable if the GUI developer could switch to another perspective that is more oriented towards Java development.

The RCP application makes it already quite easy to distribute the UCP tools. Copying one ZIP file that is created when the application is exported for a specific platform is all that has to be done. Still, that file is quite large (about 245 MB for the OS X version). Cleaning up the product definition for the RCP application could reduce the time that is required to download, copy or extract the UCP tools. Not all Eclipse plug-ins or features it currently references are actually required in the context of GUI development with UCP. Removing the superfluous ones could reduce the file size of the exported UCP application.

During the work on the graphical Discourse Model editor, it was discovered that the meta-model for Discourse Models does not clearly distinguish between opening and closing types of Communicative Acts. This led to possible inconsistencies that could be created when using the graphical editor in version 1 of the tools (e.g., specifying an `Answer` as an opening Communicative Act). For version 2 of the UCP tools, this was solved by modifying the generated code of the graphical editor to be more restrictive with regard to this issue. It may, however, be an even better solution to change the meta-model to clearly specify which Communicative Acts can be opening and which can be closing.

In the Application Adapter, the message objects that the `ApplicationAdapterInterface` is processing are implementations of the `IContent` interface. Each such object has a name property (which is a string) and a list of named parameters and attributes (which is a `Map` with strings as keys in the current implementation of the UCP tools). Such an approach is error-prone, as those string values need to be equal in both the specification of the Action or Notification (in the Action-Notification Model) and their usage in the Application Adapter. Introducing separate Java classes for each type of Action and Notification with statically typed properties instead of the generic parameter and attribute lists would allow the compiler to enforce type safety. With this approach the Application Adapter implementation would no longer have to rely on string comparisons in order to distinguish between different types of messages and to locate the associated message handler class. This option could be investigated in future work.

7 Conclusion

In the course of this work the tool support in UCP has been improved. Due to these improvements the GUI development process has become more intuitive, less time-consuming and easier to learn for novice GUI developers.

In order to achieve the objectives of this work, several new tools developed by the author have been added to UCP in order to automate additional aspects of the process. A "New Discourse Project" wizard was introduced that simplifies the creation and initial setup of new discourse projects. The wizard automatically generates all required files (model files, rendering rules, launch configuration etc.) and correctly links them together. The generated files already contain some elements which are common for every discourse project, such as an action for initiating the interaction in the Action-Notification Model and the corresponding Communicative Act in the Discourse Model. Furthermore, the GUI for that project can be generated successfully using the default property values in the launch configuration as created by the wizard, without the need for any manual change. The "New Discourse Project" wizard supports the GUI developer by saving many individual manual steps when creating a new discourse project, in comparison to version 1 of the UCP tools.

Another new tool introduced in this work is the Application Adapter interface stub generator. In version 1 of the UCP tools, the GUI developer had to manually create an implementation of the Application Adapter interface. In particular, Java code for handling every type of message specified in the interaction model had to be written by hand. This was a tedious and error-prone task since lots of identifiers had to be copied individually from the interaction model over to the Application Adapter implementation. The new Application Adapter interface stub generator in version 2 of the UCP tools performs a lot of this work automatically. It scans the Discourse Model and automatically creates handler classes for each specified action and notification. The logic that executes the correct message handler for a received message is also generated automatically. This saves the GUI developer a lot of work, since now only the actual implementation for each message type needs to be implemented in the individual message handlers.

Also some of the existing tools have been improved in version 2 of the UCP tools. The graphical Discourse Model editor has been updated to display fewer options in various menus, removing options that are not correct in a specific context. This results in a much simpler UI of the graphical editor, especially for some of the most common operations (such as creating Communicative Acts, Discourse Relations and links between those elements). But more importantly, this avoids making certain kinds of errors when editing the Discourse Model, which could cause GUI generation to fail later on. This makes the graphical editor easier to use, especially for GUI developers that have

little or no experience with UCP. Version 2 of the UCP tools also introduced a few new features in the graphical Discourse Model editor that make it more convenient to use. The associated agent for Discourse Relation elements can now be toggled more easily, and the `Arrange All` feature arranges the graphical elements in a more intuitive manner. Furthermore, it is no longer possible to accidentally delete the content compartments of Communicative Acts. These additions increase the usability of the graphical Discourse Model editor.

The fields in the launch configuration editor for the GUI code generation have been reorganized and are now more self-explanatory, which is especially helpful for novice users. Transformation rules can now be copied without having to manually change the ID afterwards in the Rendering Model.

The ability to export the UCP tools to a standalone Eclipse RCP application was restored, the UCP application can now again be exported for multiple platforms (e.g., Windows, Linux and OS X). The custom workbench that is part of this application provides a simplified UI when compared with the full-fledged Eclipse IDE, which allows the GUI developer to focus on his tasks more easily. The RCP application, of course, contains all new tools and improvements described in this work.

In comparison to version 1 of the UCP tools, version 2 requires much fewer steps to get any new project running. This makes it easier for new UCP users to get started developing GUIs with UCP and removes many possible sources of errors.

Literature

[CR06] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison-Wesley, 2nd edition, 2006.

[FKP+09] Jürgen Falb, Sevan Kavaldjian, Roman Popp, David Raneburger, Edin Arnautovic, and Hermann Kaindl. Fully automatic user interface generation from discourse models. In *Proceedings of the 13th International Conference on Intelligent User Interfaces (IUI '09)*, pages 475–476. ACM Press: New York, NY, 2009.

[FPR+07] Jürgen Falb, Roman Popp, Thomas Röck, Helmut Jelinek, Edin Arnautovic, and Hermann Kaindl. UI prototyping for multiple devices through specifying interaction design. In *Proceedings of the 11th IFIP TC 13 International Conference on Human-Computer Interaction (INTERACT 2007)*, pages 136–149, Rio de Janeiro, Brazil, September 2007. Springer.

[Gro09] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009.

[Kan14] Simon Badalians Gholi Kandi. Vergleich von Modell-getriebenen Entwicklungsansätzen für Graphical User Interfaces. Master's thesis, Vienna University of Technology, Faculty of Electrical Engineering and Information Technology, Institute of Computer Technology, E384, to be published 2014.

[LVM+04] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan. UsiXML: A user interface description language for context-sensitive user interfaces. In *Proceedings of the ACM AVI'2004 Workshop "Developing User Interfaces With XML: Advances on User Interface Description Languages"*, pages 55–62. ACM, 2004.

[Mar03] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt Series. Prentice Hall/Pearson Education, 2003.

[MPV11] Gerrit Meixner, Fabio Paternò, and Jean Vanderdonckt. Past, present, and future of model-based user interface development. *i-com*, 10(3):2–10, November 2011.

[Pat03] Fabio Paternó. ConcurTaskTrees: An engineered approach to model-based design of interactive systems. In *The Handbook of Task Analysis for Human-Computer Interaction*. Lawrence Erlbaum Associates, Mahwah, New Jersey, 2003.

[PKR13] Roman Popp, Hermann Kaindl, and David Raneburger. Connecting interaction models and application logic for model-driven generation of Web-based graphical user interfaces. In *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC 2013)*, 2013.

[Pop12] Roman Popp. A unified solution for service-oriented architecture and user interface generation through discourse-based communication models. Doctoral dissertation,

Vienna University of Technology, Vienna, Austria, 2012.

[PR11] Roman Popp and David Raneburger. A High-Level Agent Interaction Protocol Based on a Communication Ontology. In Christian Huemer, Thomas Setzer, Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, and Clemens Szyperski, editors, *E-Commerce and Web Technologies*, volume 85 of *Lecture Notes in Business Information Processing*, pages 233–245. Springer Berlin Heidelberg, 2011. 10.1007/978-3-642-23014-1_20.

[PRK13] Roman Popp, David Raneburger, and Hermann Kaindl. Tool support for automated multi-device GUI generation from discourse-based communication models. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive computing systems*, EICS '13, New York, NY, USA, 2013. ACM.

[PRS11] Jenny Preece, Yvonne Rogers, and Helen Sharp. *Interaction design: beyond human-computer interaction*. John Wiley & Sons, 3rd edition, 2011.

[PSS09] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact.*, 16:19:1–19:30, November 2009.

[PSS11] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. Engineering the authoring of usable service front ends. *Journal of Systems and Software*, 84(10):1806–1822, 2011.

[RKP+14] David Raneburger, Hermann Kaindl, Roman Popp, Vedran Šajatović, and Alexander Armbruster. A process for facilitating interaction design through automated GUI generation. In *SAC '14: Proceedings of the 2014 ACM symposium on Applied Computing*. ACM Press, to be published 2014.

[RPK+11] David Raneburger, Roman Popp, Hermann Kaindl, Jürgen Falb, and Dominik Ertl. Automated Generation of Device-Specific WIMP UIs: Weaving of Structural and Behavioral Models. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 41–46, New York, NY, USA, 2011. ACM.

[RSKF11] David Raneburger, Alexander Schörkhuber, Hermann Kaindl, and Jürgen Falb. UI Development Support through Model-integrity Checks in a Discourse-based Generation Framework. In *Proceedings of the First International Workshop on Combining Design and Engineering of Interactive Systems through Models and Tools (ComDeisMoto)*, 2011.

[Sch10] Alexander Schörkhuber. Integritätsprüfung von Diskursmodellen, Transformationsregeln und strukturellen Modellen von graphischen User Interfaces. Master's thesis, Technische Universität Wien, Fakultät für Elektrotechnik und Informationstechnik, Institut für Computertechnik, E384, 2010.