DISSERTATION

Automated Model-driven Generation of the Structure of WIMP User Interfaces Based on High-level Models

Submitted at the Faculty of Electrical Engineering, Vienna University of Technology in partial fulfillment of the requirements for the degree of Doctor of Technical Sciences

under supervision of

Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl
Institute number: 384
Institute of Computer Technology
Vienna University of Technology

and

Oscar Pastor, Full Professor
Research Center on Software Production Methods
Universitat Politécnica de Valencia
Spain

and

Proj.-Ass. Dipl.-Ing. Dr.techn. Jürgen Falb
Institute number: 384
Institute of Computer Technology
Vienna University of Technology
as participating assistant

by

Sevan Kavaldjian
Matr.Nr. 9848110
Böckhgasse 9/72, 1120 Wien

Vienna, May 2011 _____

Kurzfassung

Das Erstellen von grafischen Benutzungsschnittstellen ist immer noch eine zeit- und kostenintensive Aufgabe. Nichtsdestotrotz erlauben aktuelle Forschungsansätze diese aus Interaktionsmodellen zu generieren. Diese Dissertation präsentiert einen automatisierten Ansatz zur modellgetriebenen Generierung der Struktur von graphischen Benutzungsschnittstellen, WIMPs aus Interaktionsmodellen. Die benutzten Interaktionsmodelle sind Diskursmodelle, die genau genommen eine Klasse von Dialogen darstellen. Diese Modelle basieren auf Theorien der menschlichen Kommunikation und sollten daher für Menschen leichter verständlich sein als Quellcode, der Benutzungsschnittstellen implementiert. Solche Diskursmodelle zusammen mit entsprechenden Transformationsregeln besitzen genügend Sematik, um Benutzungsschnittstellen automatisch für eine Vielzahl von Endgeräten zu generieren.

Diese Dissertation präsentiert einen zweistufigen Transformationsprozess mit einem Zwischenmodell für die Struktur von Benutzungsschnittstellen. Sie konzentriert sich auf den ersten Schritt, die Transformation von Diskursmodellen zu Strukturmodellen von Benutzerschnittstellen mit Hilfe von deklarativen Transformationsregeln. Im Speziellen werden Transformationsregeln für Desktop-Benutzungsoberflächen, Touchscreens, die mit dem Finger zu bedienen sind, sowie Transformationsregeln für unterschiedliche Bildschirmgrößen dargestellt. Zusätzlich wird ein Ansatz zur Prototypengenerierung von grafischen Benutzungsschnittstellen, ausgehend von Anforderungsspezifikationen mit Hilfe von prozeduralen Tranformationen, vorgestellt. Es werden Transformationen, ausgehend von der Anforderungsspezifikation über die Benutzungsschnittstellenspezifikation bis hin zum grafischen Benutzungsschnittstellenprototypen gezeigt und umgekehrt. Darüberhinaus wird ein Ansatz zur automatischen Generierung von inversen prozeduralen Transformationsregeln vorgestellt. Des Weiteren sind in dieser Dissertation die zur Transformation benutzten deklarativ dargestellte Regeln zum Vergleich ebenfalls in der prozeduralen Sprache implementiert und die Unterschiede näher erläutert. Eine Feasibility-Studie wird vorgestellt, welche die Anwendbarkeit des auf Diskursen basierenden Ansatzes für die Generierung von vorgegebenen Touchscreen-Benutzungsschnittstellen, die mit dem Finger zu bedienen sind, für Einkaufswagenroboter zeigt. Es werden die dabei bewältigten Problemstellungen während der Generierung der Benutzungsschnittstelle gezeigt, sowie die daraus gewonnenen Erkenntnisse. Abschliessend wird die vorliegende Arbeit anderen Arbeiten gegenübergestellt.

Abstract

User-interface design is still a time consuming and expensive task to do, but recent advances allow generating them from interaction design models. This doctoral dissertation presents an automated model-driven approach for generating the structure of WIMPs (Window, Icon, Menu, Pointing device) out of interaction design models like Discourse Models, more precisely models of classes of dialogues. They are based on theories of human communication and should, therefore, be more understandable to humans than source code implementing user interfaces. Such Discourse Models together with transformation rules contain enough semantics to generate WIMP UIs automatically for multiple devices.

This doctoral dissertation presents a two-step transformation approach with an intermediate UI model. It concentrates on a major part of the first step, transforming Discourse Models to Structural UI Models using declarative transformation rules. In particular, transformations for desktop UIs and finger-based touchscreen UIs as well as for different screen sizes are presented. Additionally, an approach for GUI prototype generation based on requirements specifications with procedural transformation rules is introduced. It presents transformations from requirements specification via UI specification to UI prototypes and vice versa as well as an approach to automatically derive inverse procedural transformation rules. Furthermore, this work compares declaratively represented rules used for transformation, which have been implemented in a procedural language as well, for comparison with each other, and explains the respective differences. A feasibility study is presented to show the applicability of the discourse-based approach for the generation of the GUI for a given finger-based touchscreen GUI design of a robot trolley. It explains the challenges tackled during the touchscreen GUI generation and the lessons learned in the course of the application. Finally this work is put into context with related work.

Acknowledgements

First of all, I would like to thank Professor Hermann Kaindl, my supervisor, who has been a great support while writing my thesis and has always provided me with valuable pieces of advise and hints especially when I seemed to lose track. He has always been patient and helpful when I needed support or further ideas to improve my doctoral dissertation. He was a great supervisor especially when I had to deal with complex questions and problems as he supported me with his profound and highly sophisticated knowledge of Software Engineering. I have to thank him very much for motivating me and encourage me in writing this thesis.

I also have to thank my second supervisor, Professor Oscar Pastor, who enabled me to put down my own ideas and inspired me with fruitful discussions.

Furthermore I would like to thank Dr. Jürgen Falb, who always had an open ear, most valuable comments and was a great support when it comes to answer difficult questions throughout the whole thesis writing process. The same applies to Dr. Chistian Bogdan, who was especially supportive when drafting the concept of this thesis especially regarding the topic Human-Computer Interaction.

I also have to thank our research team of the Department of Computer Technology, Dr. Edin Arnautović, Dominik Ertl, Roman Popp, David Raneburger and Alexander Szép who have been most supportive when developing the concept of this thesis.

Finally I would like to thank my parents, Edna and Haik, who have always supported me and motivated me not to give up and looked after me especially in times when writing this thesis was not so easy. They have taught me that fairly everything is possible if only you stay motivated and not lose the aim out of sight. Last but not least I would like to thank my carrying girlfriend Nadja for having so much patience and giving me the energy to keep on working on my doctoral dissertation.

Vienna, May 2011 Sevan Kavaldjian

Table of Contents

List of Figures

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
ARES	Automatic Round trip Engineering System
ATL	ATLAS Transformation Language
AUI	Abstract User Interface
CIM	Computing Independent Model
CSS	Cascading Style Sheets
CTT	Concurrent TaskTrees
CUI	Concrete User Interface
DPI	Dots Per Inch
DTL	Discourse Transformation Language
EMF	Eclipse Modeling Framework
FUI	Final User Interface
GUI	Graphical User Interface
HCI	Human Computer Interaction
JET	Java Emitter Templates
KBS	Knowledge-based System
LHS	Left Hand Side
MDA	Model Driven Architecture
MDD	Model Driven Development
MDSD	Model Driven Software Development
MDUID	Model-Driven User Interface Development
MOF	Meta Object Facility
MOLA	MOdel transformation LAnguage
OCL	Object Constraints Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model

RHS	Right Hand Side
RSL	Requirements Specification Language
RST	Rhetorical Structure Theory
UI	User Interface
UIDL	User Interface Description Language
UML	Unified Modeling Language
UsiXML	USer Interface XML
WIMP	Window Icon Menu Pointer
XML	eXtensible Markup Language
XUL	XML User Interface Language

1 Introduction

This introduction explains the motivation behind this work and the research problem tackled. In particular, it presents the research hypotheses this work is based upon. Furthermore, it provides an overview of this doctoral dissertation and briefly explains the content of each chapter.

1.1 Motivation

Every device needs a device-specific UI (user interface). In particular manual creation of GUIs (graphical user interfaces) is error prone. As a consequence, recently developed approaches for GUI development try to maximize reuse and reduce coding errors by automatically generating the user interfaces for different devices (e.g., PDA, PC, mobile phone) based on the same high-level model. This model, which is platform independent, is a key element of Model Driven Development (MDD), called Platform Independent Model (PIM) and allows modeling the interaction without having a concrete device in mind.

The driving force for research activities in the area of automatic generation of user interfaces is the ability to develop UIs more cost efficiently and faster than manually developed UIs. By allowing domain experts to model the interactions and having the UI generated automatically for several target devices, the development costs may be reduced. The code generation avoids human coding errors and may, therefore, result in improved software quality as well. Modeling the interaction between a human and a computer is intuitive for end users [BKFP08]. This motivates us to use a discourse-based approach as a starting point for UI generation.

Instead of generating UIs from simple abstractions, an interaction designer (or even an end user) has the opportunity to model discourses in the sense of dialogues (supported by a tool). From such a high-level Discourse Model[1], the goal is to automatically generate the overall structure and the "look" of a GUI, more precisely a WIMP (window, icon, menu, pointer) user interface. This automatic generation employs model transformations.

[1] Discourse Models for general human-machine and machine-machine communication have been developed in a team effort in the course of the FIT-IT OntoUCP project (No. 809254/9312, www.ontoucp.org) [FKH+06, BFK+08, BKFP08, PFA+09].

1.2 Research Problem

The research problem which this thesis addresses is how to get automatically from a specific interaction design model to a concrete user interface. In this approach, an interaction design is represented as a Discourse Model. The main idea is to apply a model-driven approach to transform a Discourse Model to a Structural UI Model which models the UI structure independently of any UI toolkit. This approach allows inheriting all the advantages of MDD to the generation of user interfaces. Figure 1.1 illustrates the problem overview.

Figure 1.1: Problem Overview

The transformation rules which transform a Discourse Model to a Structural UI Model, illustrated in Figure 1.1 with an arrow, have to be identified. Applying them to a Discourse Model creates the Structural UI Model, which is the starting point for the generation of the WIMP-UI screens in a particular target language. The WIMP-UI code generation from the Structural UI Model, which is out of scope of this doctoral dissertation, is described for Java Swing in the master thesis [Ran08]. The WIMP-UI behavior is also generated from the Discourse Model and is not part of this doctoral dissertation.

Developing the UI for the same application for a new device manually, carries a lot of options for human errors. The approach presented in Chapter 3 of this doctoral dissertation allows the process of switching to a new device to be automated. Ideally, only the device specification for the new device has to be created and the generation process has to be executed.

1.3 Thesis Statement

This research is based on the following hypotheses, presented in [Kav07]:

High-level models and transformation rules capture all necessary information for WIMP-UI screen generation: We conjecture that high-level models and transformation rules provide all necessary information to generate the WIMP-UI screens. The needed high-level models are the Discourse Model and the Domain-of-Discourse Model. The Domain-of-Discourse Model defines the objects which the Discourse Model talks about. The transformation rules define mappings between the Discourse Model and the Structural UI Model. The Structural UI Model is the input for the WIMP-UI screen code generation.

Transformation rules support the generation of the structure of WIMP-UIs for multiple devices: Starting the UI development with the specification of the Discourse Model, this approach allows the generation of the structure of WIMP-UI for multiple devices automatically. The transformation process for transforming a Discourse Model to a Structural UI Model uses the device specification to select the appropriate transformation rules and, therefore, is able to achieve WIMP-UIs screens tailored for each device. The generation process uses an optimization

loop to achieve maximum use of the available space for the given resolution, minimum amount of navigation clicks, and minimum scrolling.

Specific transformation rules support the generation of UIs for different pointing granularities: UIs developed for pen-based touchscreens and mouse-based GUIs have to be designed differently compared to UIs used for finger-based touchscreens. For the different pointing granularity of a finger compared to a pen or mouse, the suitable input widget has to be selected. Therefore, this approach offers different transformation rules to support fine pointing granularity (pen-based touchscreen and mouse-based GUI) as well as coarse pointing granularity (finger-based touchscreen). According to the pointing granularity defined in the device specification, the appropriate transformation rules are selected for the Structural UI Model generation.

Transformation rules support the presentation of content according to purpose: The main ingredients of Discourse Models are Communicative Acts. They express the purpose of the communication. Each Communicative Act can be related to an object of the Domain-of-Discourse Model, which represents the content. During the UI generation process, the Communicative Act type (purpose) is used to present objects of the Domain of Discourse according to purpose. Depending on the Communicative Act type, the same content may be presented differently (according to purpose) to the user.

1.4 Thesis Structure

The remainder of this doctoral dissertation is organized in the following manner:

Chapter 2 provides some background on the field of model driven software development. It presents the model-to-model transformation concept. It also explains different existing model-to-model transformation approaches. Furthermore, it gives an insight in model driven user interface development.

Chapter 3 represents the core of this work and explains the model-driven WIMP-UI generation approach based on Discourse Models. The transformations from a Discourse Model to a Structural UI Model are illustrated for different platforms with examples. In particular, this chapter presents transformation rules for fine pointing granularity (e.g., mouse-based desktop UIs) and coarse pointing granularity (e.g., finger-based touchscreen UIs) as well as for different screen sizes.

Chapter 4 shows the generation of UIs starting with requirements specifications with a transformation language based on procedural rules. It presents transformations from requirements specifications via UI specifications to UI prototypes and vice versa. Furthermore, it explains an approach to automatically derive inverse transformations.

Chapter 5 compares a declarative and a procedural transformation language with each other. Specifically, it compares the transformation rules of both languages. It presents the advantages of each transformation language and gives a comparison summary.

Chapter 6 shows the feasibility study CommRob. It describes how we deploy the discourse-based UI generation approach to fully-automatically generate the GUI for a given finger-based touchscreen GUI design of a robot trolley. It explains the challenges tackled during the touchscreen GUI generation and presents the lessons learned during the deployment.

Chapter 7 puts this work into context of related research. In particular, it explains the UI generation approach with the OlivaNova model execution system, the UsiXML-based MDA-Compliant Environment for Developing UIs, and the generation of UIs based on Task Models.

Chapter 8 concludes and presents some ideas for future work.

2 Background on Model Driven Software Development

This chapter provides background information on the field of model driven software development. It presents the model-to-model transformation concept. In particular, it explains different existing model-to-model transformation approaches. Furthermore, it gives an overview in model driven development of user interfaces.

2.1 Model Driven Software Development in General

Model Driven Software Development (MDSD) [SV06] is based on the use of models. Models can be seen as (software) abstractions that give developers the possibility to effectively address concerns. The Model Driven Architecture (MDA) [MM03, MSUW04] is an example of a model driven approach. It defines a process composed of three different kinds of models:

Computing Independent Model (CIM): This model represents the requirements from a computation-independent viewpoint. It does not show any detail about the structure of the software system. The requirements of a system are modeled in a CIM. It plays a significant role in bridging the gap between domain experts on the one hand and software engineers on the other.

Platform Independent Model (PIM): This model is independent of the specific technological platform used to implement software.

Platform Specific Model (PSM): This model is independent of the implementation language. However, it already contains details for one particular platform.

Figure 2.1 illustrates the MDA process. The key idea is to start software development with the specification of higher-level models and to generate lower-level models and eventually code out of them. The arrows in Figure 2.1 represent model transformations that transform a more abstract model to a more concrete one (e.g., PIM to PSM).

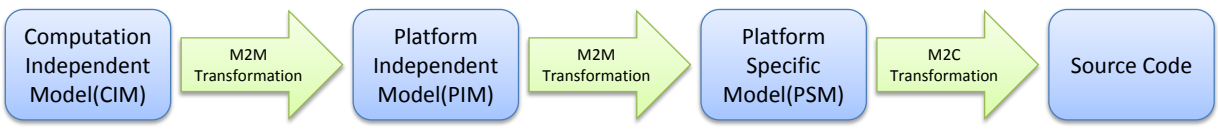


Figure 2.1: MDA Process (redrawn from [Van05])

Model transformations play a key role in MDSD. Figure 2.2 illustrates the basic concept of model-to-model transformations. It shows an example with one source model and one target model. Each source and target model conforms to a metamodel. A metamodel represents the abstract syntax of a model. Transformation rules are defined by mapping source metamodel elements to target metamodel elements. The transformation rules are executed by the transformation engine. The source model is read by the transformation engine and the corresponding target model is created. In some cases the source and target metamodels can be the same.

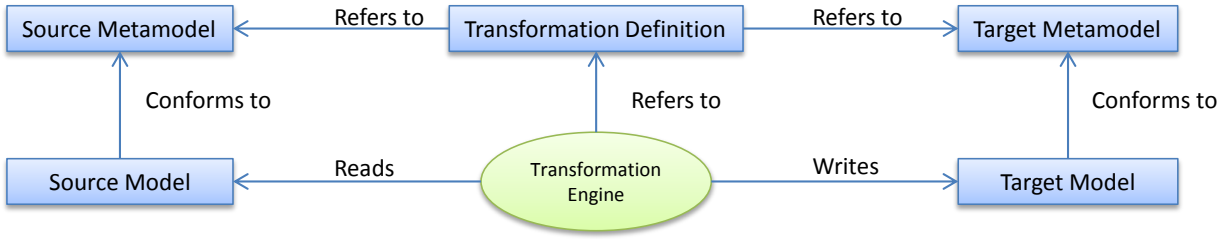


Figure 2.2: Model-to-model transformation concept (redrawn from [CH06])

2.2 Model-to-Model Transformation Approaches

There are several different model-to-model transformation approaches in the literature [CH06]:

- The direct manipulation approach
- The structure-driven approach
- The operational approach
- The template-based approach
- The relational approach
- The graph-transformation-based approach
- The hybrid approach

In the following, each of the approaches is explained according to [CH06]:

Direct manipulation approach: This approach is implemented as an object-oriented framework which can offer some minimal infrastructure to organize the transformations. It provides an internal model representation and some APIs to modify it. Many facilities like transformation rules, scheduling, tracing have to be implemented from scratch, in a programming language like Java.

Structure-driven approach: The characteristic of this approach are two distinct phases during the transformation. The first phase is responsible for creating the hierarchical structure of the target model. The second phase takes care of setting the attributes and references in the target. The framework is in charge of the scheduling and application strategy. Developers have only to write the transformation rules. The transformation language used in Chapter 3 belongs to this category.

Operational approach: This category of approaches is quite similar to direct manipulation but provides more dedicated support for model transformation. In this approach, facilities to express computations are used to extend the utilized metamodeling formalism. A typical example would be to extend a query language like the Object Constraints Language (OCL) with imperative constructs. An object-oriented programming system is composed by combining the Meta Object Facility (MOF) with such an extended executable OCL. Systems in this category are, e.g., QVT operational mappings and Kermeta.

Template-based approach: This approach uses model templates which are models with inserted metacode that calculates the variable parts of the resulting template instances. The developer has to anticipate the result of the template instantiation with the help of the model templates, which are expressed in the concrete syntax of the target language. The form of the metacode can be annotations on model elements. All annotations are part of the metalanguage and can particularly be conditions, iterations, and expressions. In most cases the used expression language in the metalanguage is OCL.

Relational approach: This approach is declarative and can be characterized by the main concept of mathematical relations. They can be interpreted as a form of constraint solving. The key principle is to use constraints to specify the relations between source and target element types. Originally such specifications are not executable. In contrast this approaches can, comparable to logic programming, give declarative constraints executable semantics. In relational approaches, predicates are used to describe the relations. Therefore, it is often implemented in logic programming with its unification-based matching, search, and backtracking. In rational approaches target elements are created implicitly, whereas in imperative direct manipulation approaches explicitly. Relational approaches are side-effect-free, support multidirectional rules and even sometimes provide backtracking. They require the separation among source and target models and thus do not offer the possibility for in-place update. Systems in this category are, e.g., QVT Relations and Kent Model Transformation Language.

Graph-transformation-based approach: The graph-transformation-based approach is based on the theoretical work on graph transformations. It operates on typed, attributed, labeled graphs which can be seen as formal representations of simplified class models. Each graph transformation rule is composed of a Left Hand Side (LHS) and Right Hand Side (RHS) graph pattern. The LHS pattern is matched in the source model, whereas the RHS pattern is created in the target model. In many cases, LHS patterns additionally contain conditions. Additional logic is needed to calculate target attribute values like element names. Developers have the choice to transform graph patterns in the concrete syntax of their source or target language or in the MOF abstract syntax. The concrete syntax has the advantage to be more familiar to developers used to work with a modeling language compared to the abstract syntax. However, the abstract syntax can be used to allow a default transformation that will work for each metamodel. This can be useful when no specific concrete syntax is available. Systems in this category are, e.g., MOLA and Fujaba. MOLA is used in Chapter 4 as the transformation language.

Hybrid approach: This approach uses different techniques from the other model-to-model approach categories. They can be combined in different ways. Either they can be combined as separate components or at the level of individual rules. A declarative rule has an LHS (source pattern) composed of a set of syntactically typed variables, which have an optional OCL constraint as a filter and an RHS (target pattern) composed of a set of variables and a declarative logic to assign values to the attributes of the target elements. A hybrid rule has a block of imperative logic as a complement to the source or target patterns, which is executed after the target pattern has

been created. However, an imperative rule is composed of a name, an imperative block, and a set of formal parameters, though no patterns are declared. An example of a hybrid approach is QVT, which has three separate components, Relations, Operational mappings and Core. Combinations on the rule level, for example, are ATL and YATL.

2.3 Model-Driven User Interface Development

Model-Driven User Interface Development (MDUID) is a quite new field of research that combines techniques from model driven software development with Human Computer Interaction (HCI) concepts. The Cameleon Reference Framework [Van05, CCT+03] defines the UI development on four different levels, illustrated in Figure 2.3. An example of a model located on each of the four levels is shown.

The Task and Domain Model is the most abstract model which corresponds to the CIM in the MDA Process. It can be notated e.g., in the Concur Task Tree (CTT) notation. It is illustrated in Figure 2.3 in the blue area. It describes tasks of the user (e.g., create brand) and their temporal relationship.

The Abstract UI Model which is more concrete corresponds to the PIM. It describes the UI independently of any interaction modality and platform. It contains abstract containers and abstract interaction components. In Figure 2.3 the yellow area shows an Abstract UI Model. It describes the UI elements in an abstract way (e.g., element for creating a brand).

The Concrete UI Model corresponds to the PSM. It describes the UI for a particular interaction modality (e.g., GUI) but is still independent of the UI-toolkit. Each platform (e.g., smartphone, desktop PC) has its specific Concrete UI Model. This model already contains the widgets that are supported by the particular platform. An example of a Concrete UI Model is illustrated in Figure 2.3 in the orange area, which shows the widget structure of the GUI, which is composed of four textboxes, each having a label next to it, and a button below them.

The Final UI Model corresponds to the code in the MDA Process. It represents the UI code that is created based on the previous levels. It is illustrated in Figure 2.3 in the pink area. It shows the final resulting GUI screen.

If we add model transformations to get from one model of the Cameleon Reference Framework to another, it can be interpreted as the MDA Process tailored to User Interfaces development.

The most important existing approaches for model driven user interface development are explained in Chapter 7 in the context of the related work.

2.3.1 Model-Driven GUI Structure and Behavior Generation

Every GUI has a structure and a behavior. Therefore, approaches for model-driven development of GUIs have to generate the structure as well as the behavior on some level of abstraction starting from the task and domain level model. Two different approaches are possible with each of them having pros and cons. The first one illustrated in Figure 2.4 generates separate models for the GUI structure and behavior starting from the task and domain level model. From both generated models the Final UI is generated in a second step. The approach presented in Chapter 3 of this doctoral dissertation is based on this model-driven GUI generation process. However this doctoral

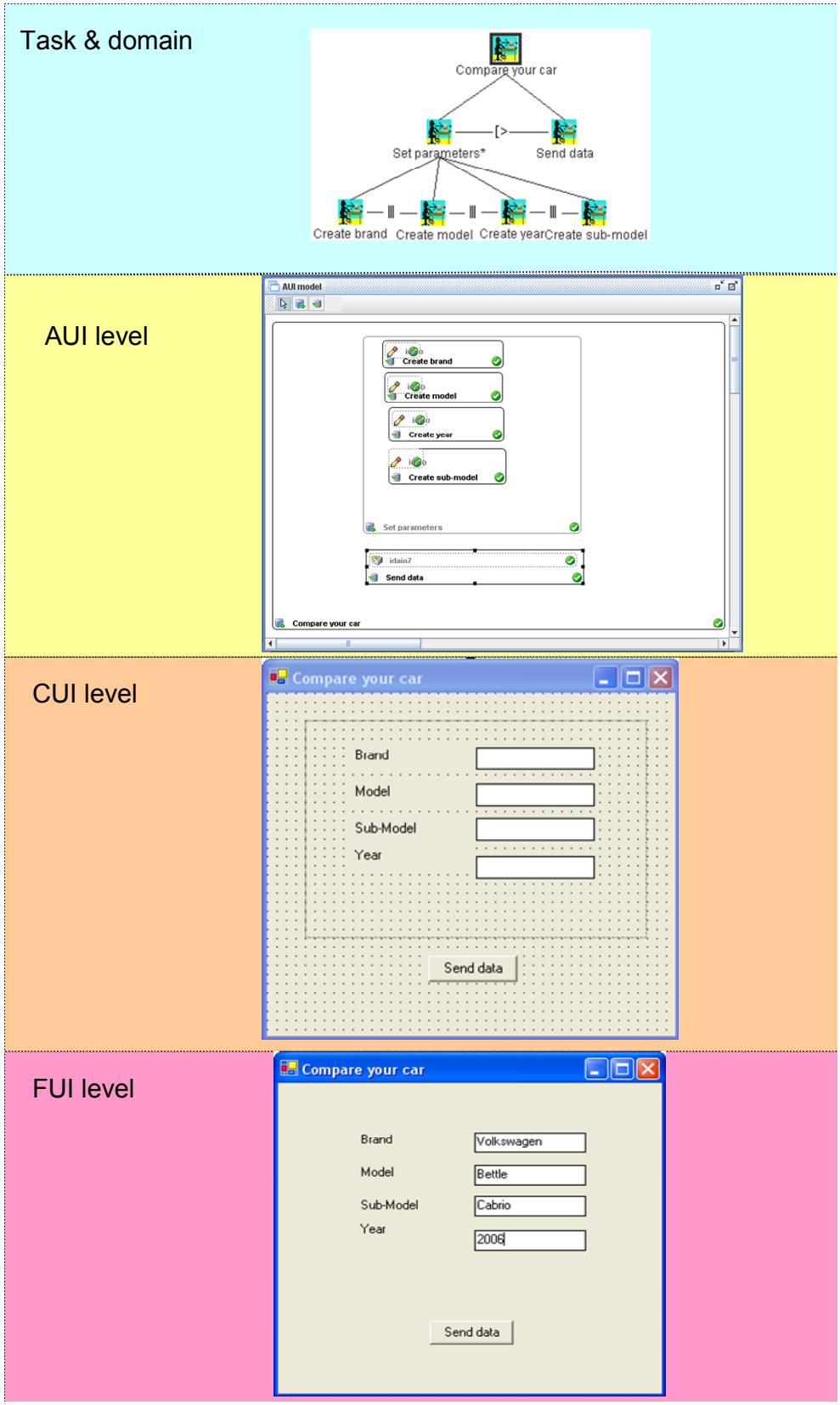


Figure 2.3: The Cameleon Reference Framework (copied from [Van08])

dissertation focuses only on the generation of the GUI structure, which is based on the model-to-model transformations from the task and domain level model to the model capturing the UI structure illustrated with the arrow in the top left of Figure 2.4. The GUI behavior generation is out of scope of this dissertation. The process of combining the GUI behavior and structure model to the Final UI is explained in [RPK+11a].

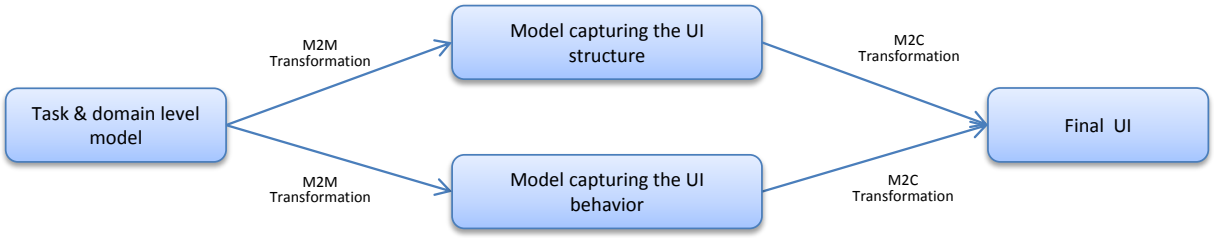


Figure 2.4: UI generation with separated structure and behavior model

The positive and negative aspects of GUI generation with separated structure and behavior are:

- Pro: Separation of concerns regarding the behavior and the structure of the GUI
- Con: GUI behavior and structure have to fit together

In contrast to Figure 2.4, Figure 2.5 illustrates a GUI generation process which generates one model that captures the structure as well as the behavior of the GUI. Thus, there is no separation of concerns. The Final UI is generated straight-forward from the model capturing the structure and behavior of the GUI. The advantage of this approach is that there is no risk of inconsistencies because the complete Final UI is generated out of one model. The disadvantage is that there is no separation of concerns regarding the behavior and the structure of the GUI.

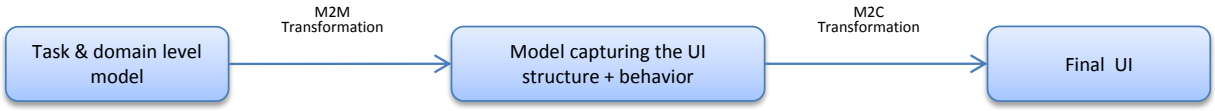


Figure 2.5: UI generation process with combined structure and behavior model

2.3.2 Model-Driven Fully- vs. Semi-Automatic UI Generation

The model-driven UI generation process can be fully-automatic or semi-automatic. Fully-automatic means that after the generation process has been started no manual modification is needed in order to generate the Final UI. In contrast, a semi-automatic UI generation process needs manual steps during the transformation process as illustrated in Figure 2.6. The manual step can be e.g., a certain step in the transformation process that has to be done manually, selecting the transformation rule to apply or to modify the UI structure on a certain level of abstraction (AUI or CUI).

MDUID implies a conflict of interest. On the one hand, the UI generation process should be as automated as possible, on the other hand it should be possible to influence the result of the generation process, semi-automated. Fully-automatic UI generation provides the best results when requirements are only given on the task and domain level. Thus, it has certain drawbacks when

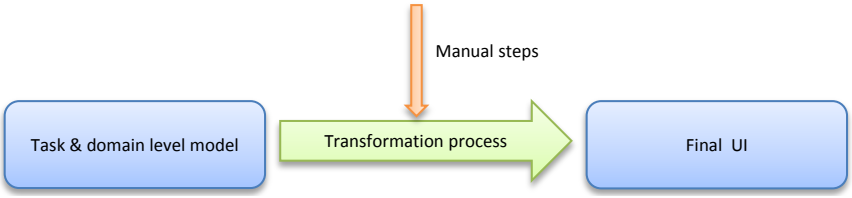


Figure 2.6: Semi-automatic UI generation process

a UI has to be generated according to a predefined UI design because it lacks the possibility to easily influence the resulting Final UI. The approach presented in Chapter 3 of this doctoral dissertation is fully-automatic. In Chapter 6 it is explained how the fully-automatic approach of Chapter 3 deals with a predefined UI design. A semi-automatic UI generation process has the advantage to influence the generation result. Thus, it better supports the generation of UIs based on a given UI design. An idea of how the approach presented in Chapter 3 can be extended to support semi-automatic UI generation is presented in [Ran10].

The ideal MDUID environment supports both fully-automatic and semi-automatic UI generation. Depending on the given requirements the appropriate generation process is chosen.

3 Automated Discourse-based WIMP UI Structure Generation

This chapter contains the main contribution of this doctoral dissertation. It presents the automated model-driven WIMP UI structure generation approach based on Discourse Models. It focuses on the transformations from Discourse Model to Structural UI Model. It starts with the transformation approach used to generate the WIMP UI structure. Then it presents the input model of the transformation, the Discourse Model and the Domain of Discourse Model. Afterwards the output model of the transformation is presented, the Structural UI Model. The applied transformation process is explained. Subsequently the core of this work, the transformation rules to transform a Discourse Model to a Structural UI Model are presented. In particular, the transformation rules for general purpose GUI [KBFK08], content transformation rules specific to intention [KFK09], finger-based touchscreen specific transformation rules [KRF+09] and transformation rules for different screen sizes [KRP+10] are presented. Finally, the model-to-code transformation is explained briefly, which is covered in detail in the diploma thesis [Ran08].

3.1 The Transformation Approach

A user interface generation process [KFK09] that transforms such Discourse Models into WIMP-based graphical user interfaces (Windows, Icons, Menu and Pointers) has been developed. The basic user interface generation process is illustrated in Figure 3.1 and consists of two steps.

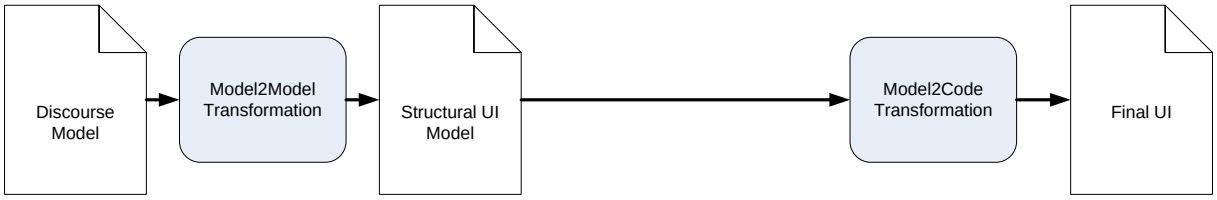


Figure 3.1: The Basic Transformation Process

First the Structural UI Model is generated, using the Discourse Model and a device specification as input. This Structural UI Model specifies the widget hierarchy of the UI. The Structural UI Model is still independent of the GUI toolkit (in our case Java Swing) used to display the screens, but it depends already on the target device, e.g., a touchscreen. Default values for metric widget

sizes are chosen according to the specified metric screen size of the device. Device properties like screen resolution, etc., are taken into account either directly during the Structural UI Model generation or by the code generator. In the second step, a code generator translates the Structural UI Model into GUI toolkit specific source code.

The separation of the rendering process in two steps has the major advantage that the generated Structural UI Model is still platform independent and can be translated into several different GUI-toolkit languages in a second step. Another advantage is the possibility of influencing the screen design through modifying the generated Structural UI Model, before triggering the actual code generation, which is out of scope of this doctoral dissertation. This could lead to a more satisfying Final UI.

3.2 Transformation Input - Discourse Model

The starting point for the automatic WIMP UI structure generation is a Discourse Model, see [FKH+06, BFK+08, BKFP08, PFA+09]. Such a Discourse Model serves as an interaction design on a high level of abstraction and is based on concepts of human language theories. It is largely a declarative model and uses a self-defined Domain Specific Language (DSL) for specifying the classes of possible dialogues or interactions between the human and the machine. The abstract syntax of the DSL is based on the conceptual metamodel shown in Figure 3.2, which illustrates the concepts used. It has the following key ingredients:

- *Communicative Acts* as derived from speech acts [Sea69] carrying *propositional content*,
- *Adjacency Pairs* adopted from Conversation Analysis [LFG90], and
- Discourse Relations, which are further specialized into *RST Relations* inherited from Rhetorical Structure Theory (RST) [MT88] and Procedural Constructs.

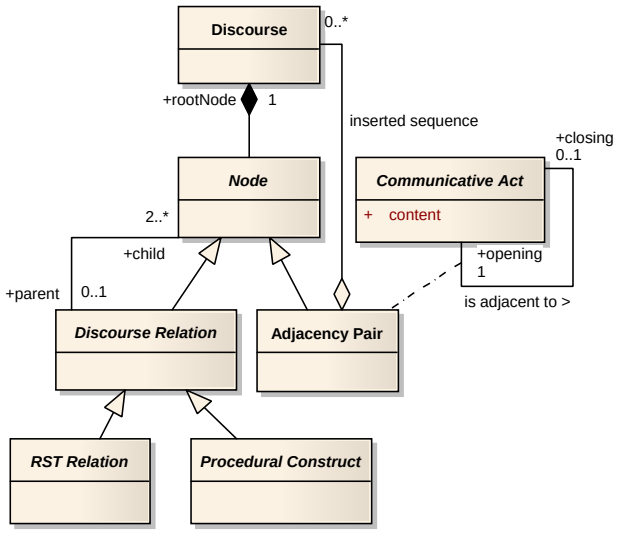


Figure 3.2: Conceptual discourse metamodel

Discourse Relations relate Nodes which can be Adjacency Pairs or other Discourse Relations, thus building up the hierarchical structure of the discourse. Adjacency Pairs can contain embedded dialogues, called inserted sequences, like clarification dialogues which may become necessary before a communication party is able to answer a question, for example due to unreliable speech input in multi-modal user interfaces. Thus, Adjacency Pairs are modeled in our metamodel as association classes.

Communicative Acts represent basic units of language communication. Thus, any communication can be seen as enacting of Communicative Acts, acts such as making statements, giving commands, asking questions and so on. The Communicative Acts indicate the intention of the interaction, e.g., asking a *question* or issuing a *request* and carries propositional content referring to elements of the domain of discourse. The domain of discourse specifies the elements the discourse can talk about.

Figure 3.3 shows a selection of the most important Communicative Acts used in this approach. Two corresponding Communicative Acts, like *Offer* and *Accept*, form a sequence, which is called *Adjacency Pair* to define the turn-taking and thus the order of the utterances. The Adjacency Pairs build up the dialogue structure. In the special case of just informing the other dialogue partner without making a dialogue turn, the Adjacency Pair degrades to an Adjacency Pair with only an opening Communicative Act as shown in the satellite branch of the Background Relation on the right in Figure 3.20.

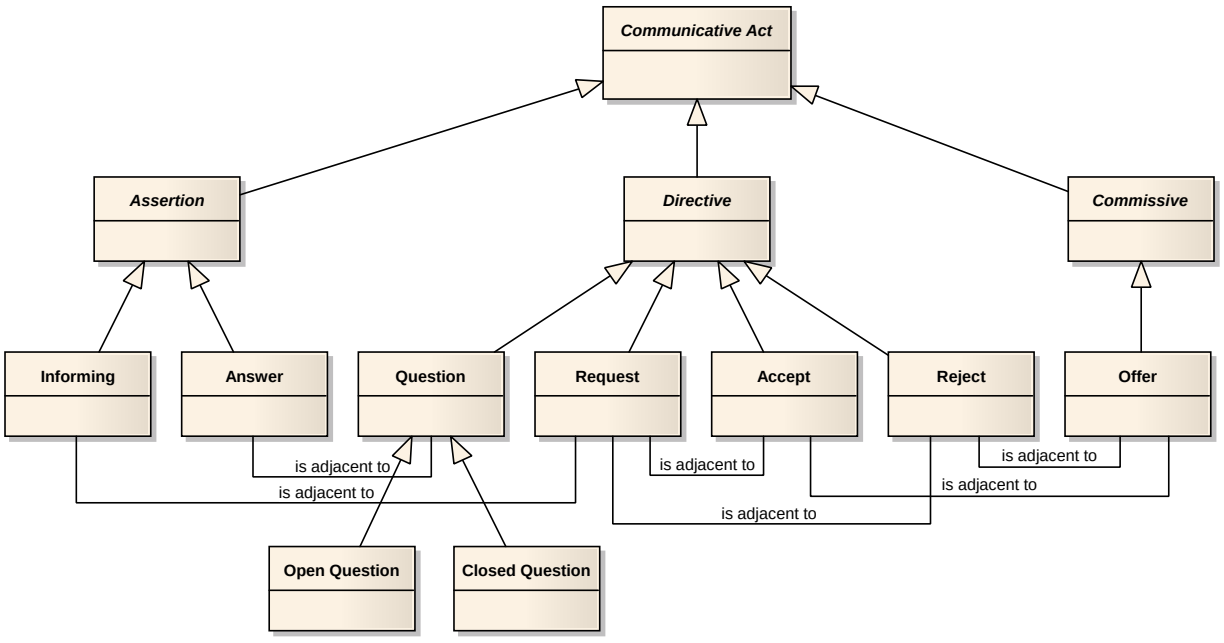


Figure 3.3: Selection of Communicative Act taxonomy

Figure 3.4 shows a selection of the most important RST Relations and Procedural Constructs used in this approach. RST Relations specify relationships among text portions and associated constraints and effects. The relationships in a text are organized in a tree structure, where the rhetorical relations are associated with non-leaf nodes, and text portions with leaf nodes. In this approach we make use of RST Relations for linking adjacency pairs and further structures made up of RST relations with each other. They describe a subject-matter relationship between the branches they relate and can be divided in two groups. RST SingleNucleusRelations having one

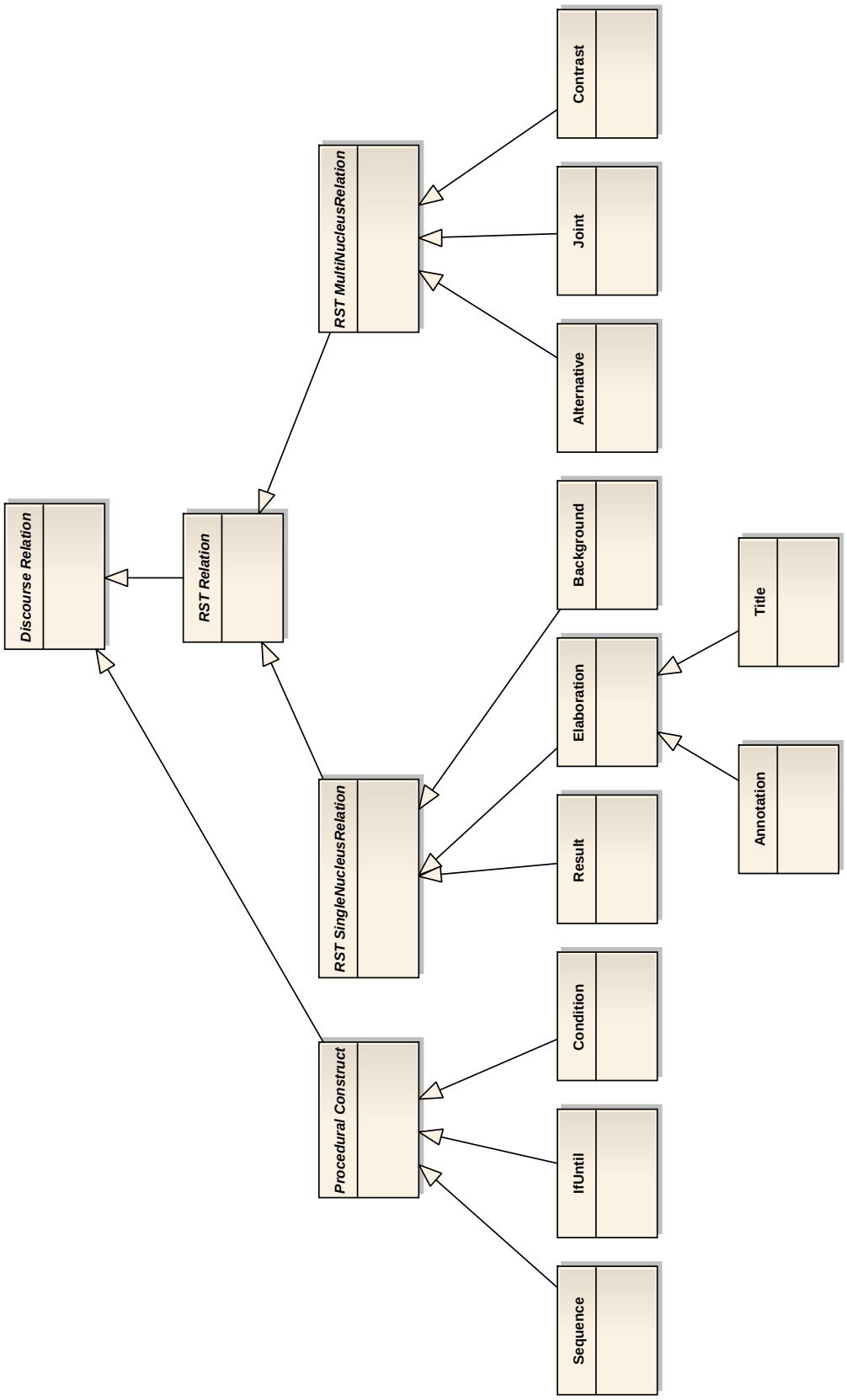


Figure 3.4: Selection of Discourse Relation taxonomy

nucleus and one satellite branch (e.g., *Background*) and RST MultiNucleusRelations having only multiple nucleus branches (e.g., *Joint*). The *Joint* Relation is used to group Communicative Acts of the same type. No presentation order is presumed. The *Background* Relation is used to express that the *satellite* branch contains background information related to the *nucleus* branch.

Procedural Constructs,e.g., *IfUntil* provide means to express a particular order between branches of the discourse tree, to specify repetition of a branch and to specify conditional execution of different branches. Thus, Procedural Constructs add control structures to our discourse trees that are more complex than usual if-then-else or repeat-until constructs in typical procedural programming languages. When operationalizing the discourse tree, these Procedural Constructs also determine which information cannot be presented together on one screen of a GUI. The *IfUntil* Procedural Construct repeats the tree branch until a condition is fulfilled to execute the then branch. A repeat condition restricts the number of allowed repetitions until the else branch is uttered.

3.3 Transformation Output - Structural UI Model

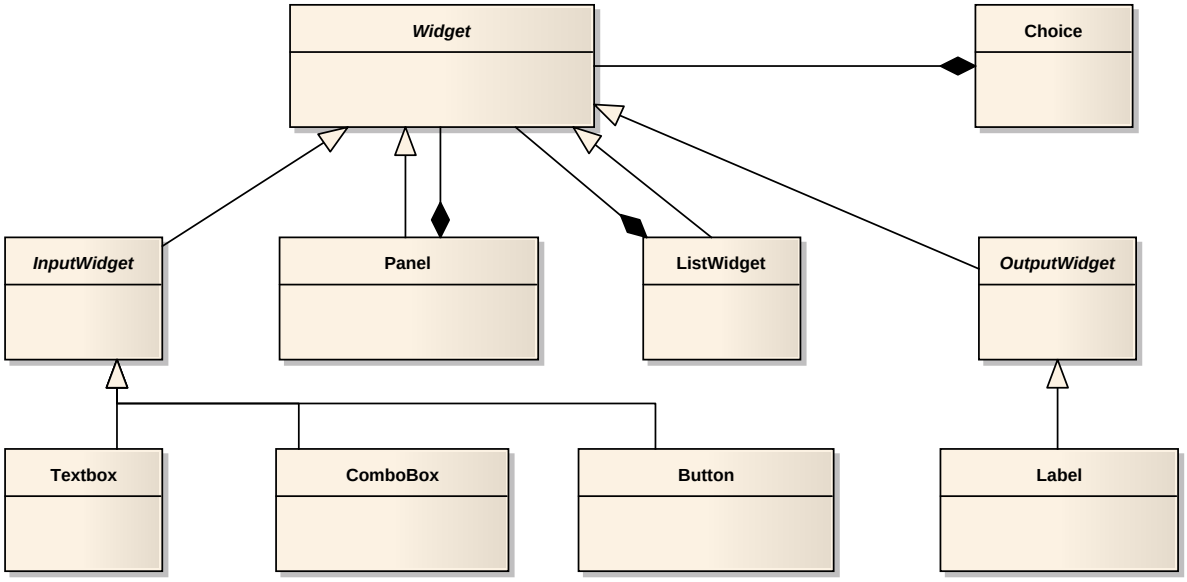


Figure 3.5: Part of conceptual metamodel of Structural UI Models

The Structural UI Model is basically a tree representing the UI structure independently of any toolkit (e.g., Web, Java Swing, etc.). It is not completely independent of the target device, however, since the device's real-estate is taken into account for building up the UI structure. Still, our Structural UI Model is completely independent of the considered UI toolkit. This tree structure will be transformed to a toolkit-specific Final UI. The concepts used in such a Structural UI model are specified in the Structural UI metamodel shown conceptually in Figure 3.5. It only shows the parts that are important to understand the examples illustrated later. The most important concept of the metamodel is the *Widget* class. It is specialized into four functional categories: *OutputWidget*s, *InputWidget*s, *ListWidget*s and *Panel*s. The *OutputWidget*s present information to the user in different ways like text and images and the *InputWidget*s gather information from

the user. Nevertheless, *InputWidget*s also convey information to the user like defaults, current values and type and quantity of required information.

In the Structural UI Model, a complete tree or subtree with a *Panel* as its root element represents a presentation unit. Hence, a complete Structural UI Model can, in general, be a forest consisting of possibly several trees. Trees in the Structural UI Model that are alternatives, i.e., trees on the same level resulting from discourse partitioning or Joint relations, will be linked in the Structural UI Model via a Choice element as shown in the metamodel in Figure 3.5. The Choice element is used to specify alternative presentation units, which can be used to fill in the same space.

3.4 The Discourse to Structural UI Model Transformation Process

Transforming Discourse Models to Structural UI Models [KBFK08], by applying model-to-model transformations to Discourse Model elements, potentially results in a different Structural UI Model for each device as illustrated in Figure 3.6. The resulting Structural UI Model represents the user interface's widgets and their structure, but still abstracts from details of the Final UI. A common UI description language (e.g., UsiXML[1]) is not used because the runtime environment is based on the exchange of Communicative Acts.

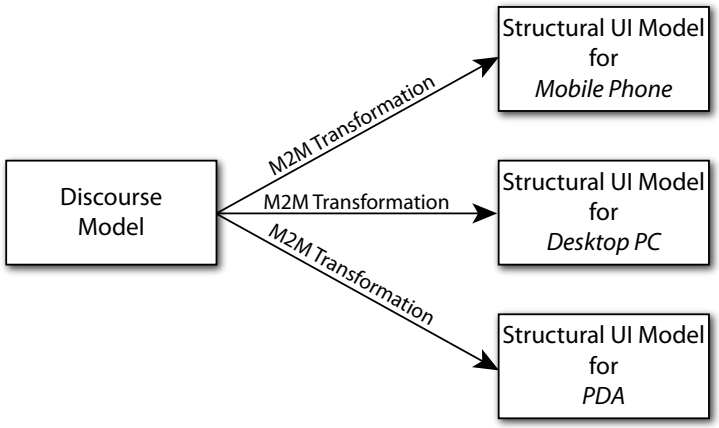


Figure 3.6: The model-to-model transformation step

Figure 3.7 illustrates that the transformation of one presentation unit is fulfilled by mapping elements of the discourse metamodel to elements of the Structural UI metamodel. Both metamodels are based on the Ecore[2] meta-metamodel. Transformation languages like ATL[3] (ATLAS Transformation Language) or MOLA[4] (MOdel transformation LAnguage), which is used in this doctoral dissertation to specify procedural transformation rules in Chapter 4 and 5, also support the same transformation process, but these approaches lack conflict resolution strategies, requiring rules to be carefully designed in a way that only one rule matches a model element at any time. This is inappropriate for GUI rendering, since one wants to provide some general rendering rules and some more specific ones matching the same element with the specific ones taking precedence over

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

[2] Essential MOF like core meta model of the Eclipse Modeling Framework (`http://www.eclipse.org/emf/`)

[3] `http://www.eclipse.org/m2m/atl/`

[4] `http://mola.mii.lu.lv/`

the general ones. At the same time a state machine is generated from the Discourse Model which controls the sending and receiving of Communicative Acts. This latter generation is beyond the scope of this doctoral dissertation, however.

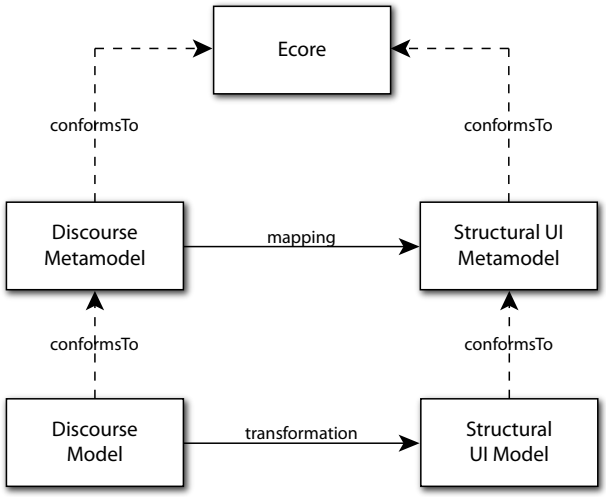


Figure 3.7: The Discourse Model to Structural UI Model transformation process

The Discourse Transformation Language (DTL) has a rule-based engine that is based on the Ecore[5] model. The metamodels for Discourse Models and Structural UI Models are instances of the Ecore model and thus provide common model navigation in the source and target model and linking between elements of both models. This feature is used to add traceability links from widgets in the Structural UI Model to the original elements in the Discourse Model. Thus, the original information is also available during the final screen generation process.

The transformation engine iterates over all elements of a Discourse Model and first checks which rules trigger for the currently processed element. For a rule to trigger, two conditions must hold:

- the discourse pattern in the rule must match the corresponding part in the Discourse Model, and
- specified rule constraints must match the device properties we want to render for (e.g., screen real estate).

The transformation engine gets the discourse and the domain-of-discourse models as input and transforms them into a Structural UI Model. This transformation takes device properties into account. The transformation engine performs a model-to-model transformation based on rules. Such a transformation rule can state, for example, that each "Informing" communicative act found in the Discourse Model has to be transformed to a panel containing a label widget for each domain-of-discourse object referred to by the communicative act's propositional content.

Our model-to-model transformation process consists of two interleaved transformation steps:

1. The first step applies rules to Discourse Model elements that generate an overall UI structure by use of pattern matching. These rules generate abstract widgets like labels for headings and placeholders for data of the propositional content. They also associate parts of the propositional content with the generated placeholders.

[5]see `http://www.eclipse.org/modeling/emf/` for the Eclipse Modeling Framework (EMF) specification.

2. The second step executes content transformation rules within the context of the rules of the first step. This embedding allows the selection of abstract widgets for the resulting structural UI depending on the content type, the content's referring communicative act type and the current context the communicative act is embedded in as defined by the enclosing rule.

3.5 Transformation Rules

After having introduced the general transformation principles as well as the input and the output model of the transformation, it is concentrated on the transformation rules for mapping a Discourse Model to a Structural UI Model. They are specific to certain structural patterns occurring in the Discourse Models. Discourse Model excerpts are used as examples to illustrate the impact of different kinds of rules. However no Discourse Model was found that covers all desired aspects. Thus, the Discourse Model excerpts had to be taken from different Discourse Models to illustrate different aspects of the transformation rules.

The transformation rules have been developed starting with the specification of Discourse Model examples, e.g., online shop, flight booking. First the desired GUI for a Discourse Model example was sketched on paper. Then the Structural UI Model according to the sketched GUI structure was created manually. Finally the transformation rules that create the desired Structural UI Model parts have been derived considering the source Discourse Model and the expected Structural UI Model. Different examples have been used to achieve generalized rules that are suitable for many cases.

3.5.1 General Purpose GUI Transformation Rules

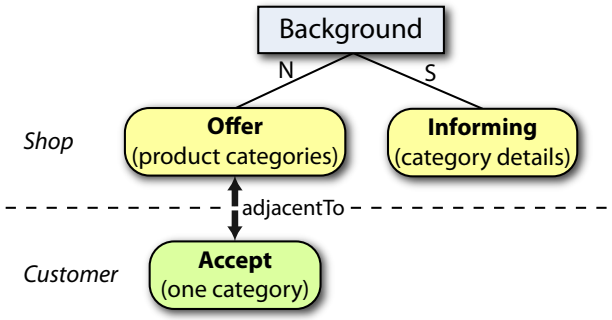


Figure 3.8: Subtree of an online shop discourse

Figure 3.8 shows a small part of an online shop Discourse Model, which we use as an example to illustrate some general purpose transformation rules. The example describes an interaction between the user and the online shop with the purpose of demanding the customer to select one product category and supporting her with background information to ease her choice. The *nucleus* branch *N* of the *Background* relation conveys the main interaction sequence. The online shop system *offers* a list of *product categories* to the user. The user *accept*s one of them. During the offering process the *satellite* branch *S* provides background information about the product categories to the user. This part of an online shop Discourse Model gets transformed to the Structural UI Model shown in Figure 3.9 by applying the rules *Background*, *Adjacency Pair*, *Offer-Accept* and *Informing* to the corresponding Discourse Model elements in the listed order.

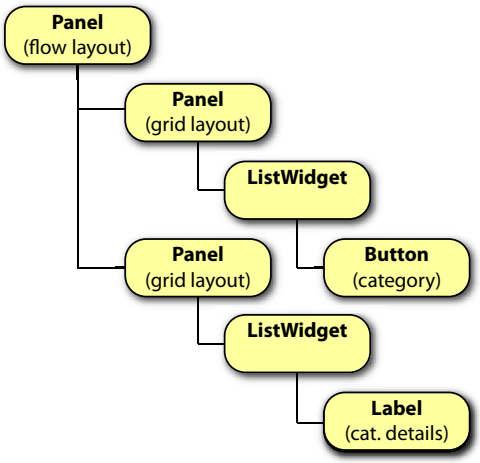


Figure 3.9: Resulting Structural UI Model

The main contribution of this doctoral dissertation is how to transfer models as exemplified in Figure 3.8 to Structural UI Models at the abstract widget level as in Figure 3.9. In particular, it means a transformation from a mainly declarative model of a discourse to the toolkit-independent structure of a user interface.

Figure 3.10 shows the generated GUI screen resulting from the Structural UI Model in Figure 3.9 corresponding to the Discourse Model excerpt in Figure 3.8. The labels from the *Informing* Communicative Act are rendered next to (right side) the buttons of the *Offer* Communicative Act.

Figure 3.10: Concrete UI (Screenshot)

Background Rule: Figure 3.11 shows a rule specific to the Background RST relation. The satellite is rendered on the right side of the presentation unit, while the nucleus occupies the left area, as an “aside”. In accordance to the rule above, the “most nuclear part” takes the interface space that is of highest surface and most central to the user focus. Following this principle further, the layout management of the presentation unit will always give precedence to the “nuclear" side, e.g., when the window is resized by the user. This rule is used to generate the basic tree structure of Figure 3.9, i.e., this rules generates the root panel and places the transformation results of the pre-rendering of the nucleus and satellite subtrees next to each other by a flow layout manager.

The Background Rule can also be localized (adapted), e.g., for cultures that write from right to left, where it may be more suitable to place the satellite at the left side. A system-wise style configuration can also render light backgrounds to the top and to the left, like it is, e.g., customary when the concrete user interface will be an HTML page. However, even there the space allocation and re-allocation in case of resizing will prioritize the nuclear part. In Chapter 5 in Figure 5.2 this rule is illustrated specified in DTL.

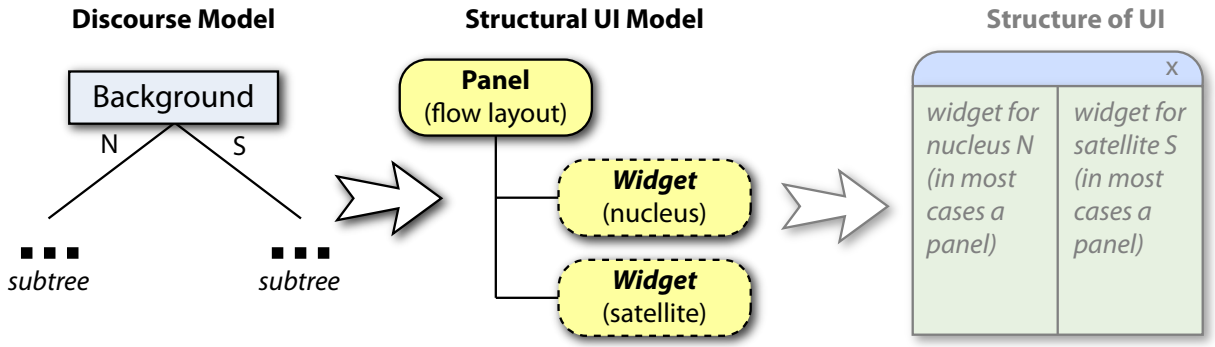


Figure 3.11: Background Rule.

Adjacency Pair Rule: Each Adjacency Pair is transformed to a *Panel* element of the Structural UI Model containing widgets according to the related Communicative Acts. In this example, the first panel on the second level in Figure 3.9 results from the application of the Adjacency Pair Rule to the Offer-Accept adjacency pair. In Chapter 5 in Figure 5.4 this rule is illustrated specified in DTL.

Offer-Accept Rule: Each *Offer-Accept* adjacency pair is transformed either to a *Button* or to a *ListWidget* element containing *Button*s, depending on the cardinality of the content offered. Because our example online shop can offer more than one product category, the *ListWidget* element is needed to model an undefined number of categories. Since the acceptance of an *Offer* requires a user action, a *Button* element is embedded in the *ListWidget* in Figure 3.9. As a result, the subtree of the *ListWidget* is repeated according to the actual number of product categories in the Final UI. In Chapter 5 in Figure 5.6 this rule is illustrated specified in DTL.

Informing Rule: Each *Informing* communicative act is transformed either to a *Label* element or to a *ListWidget* element containing a *Label* element, depending on the cardinality of the content. This rule assumes that the information will be forwarded in textual form, otherwise, e.g., a *PictureBox* or *AudioPlayer* element will be used. In the online shop example, information is conveyed for each product category and, therefore, a *ListWidget* containing *Label*s is generated.

In the following for each Discourse Model element, which is not covered yet and by the following Discourse Model excerpt examples, one general purpose rule is presented that creates one possible target pattern. However in many cases there are different possibilities to render a source pattern and therefore multiple rules exist for each source pattern. The transformation rules are named according to the source element they match.

Transformation rules for Communicative Acts:

Open Question-Answer Rule: The Open Question-Answer Rule matches an Adjacency Pair relating an open question and an answer and transforms it to a panel containing a label for the question text, input and output widget placeholders for the content, and a submit button for

submitting the answer. The rule also assigns all attributes of the open question's propositional content type to both placeholders.

Request-Accept, Reject Rule: The "Request-Accept, Reject Rule" matches an Adjacency Pair relating a request with an accept and reject and transforms it to a panel containing a label for the request text, an accept button for accepting the request and a reject button for rejecting the request. This rule layouts the label on top of both buttons. The buttons are placed next to each other, the accept button on the left the reject button on the right.

In some cases it makes sense to define Adjacency Pairs where the opening Communicative Acts is send by the user and the closing by the machine. An example would be the Request-Accept, Reject Adjacency Pair with the following transformation rule:

Request-Accept, Reject Rule (Opening Communicative Act send by user): The rule matches an Adjacency Pair relating a request, which is send by the user and refers to an object of the domain of discourse, with an accept and reject and transforms it to a choice containing a panel each for request, accept and reject. The request panel contains a label for the request text, input and output widget placeholders for the content, and a submit button for submitting the request. The rule also assigns all attributes of the request's propositional content type to both placeholders. The accept panel contains a label for the headline "Request Accepted" and a label for the accept message received form the machine. The reject panel contains a label for the headline "Request Rejected" and a label for the reject massage received from the machine. In contrast to the "Request-Accept, Reject Rule" where the Opening Communicative Act is sent by the user and the UI only has to provide the possibility to accept or reject the request, this rule has to provide the possibility to send the request as well as the presentation of the acceptance or rejection.

Transformation rules for Procedural Constructs:

The Procedural Constructs have different behavioral semantics but the same influence on the UI structure. They result in the unique presentation of one of the their branches at the same time, which is modeled in the target pattern of the rule with a choice element as the root.

Transformation rules containing choice as the root element of the target pattern cannot be layouted because the containing elements will not be displayed at the same time and, therefore, will not have to share the screen space.

Sequence Rule: The Sequence Rule matches the Sequence Procedural Construct and transforms it to a choice.

IfUntil Rule: The IfUntil Rule matches the IfUntil Procedural Construct and transforms it to a choice containing a panel for each of the three branches of the IfUntil.

Condition Rule: The Conition Rule matches the Condition Procedural Construct and transforms it to a choice containing a panel for each of the two branches of the condition.

Transformation rules for RST MultiNucleusRelations:

Transformation rules for RST MultiNucleusRelations can not be pre-layouted in the rule due to the undefined number of nucleus branches. It has to rely on the automatic layouting, which will try to put the nucleus branches next to each other if possible.

Joint Rule: The Joint Rule matches the Joint relation and transforms it to a panel. All nucleus branches of the Joint relation are mapped to the panel.

Alternative Rule: The Alternative Rule matches the Alternative relation and transforms it to a choice.

Contrast Rule: The contrast Rule matches the Contrast relation and transforms it to a choice.

In some cases it makes sense to define Discourse Relations which are evaluated by the user instead of the machine. An example for a rule for such a Discourse Relation is given in the following:

Alternative Rule (evaluated by the user): This rule matches the Alternative relation which is assigned to the user and transforms it to a panel. All Alternative relation branches will be displayed at the same time to give the user the possibility to choose from the alternatives and layouted by the automatic layouter.

Transformation rules for RST SingleNucleusRelations:

Elaboration Rule: The Elaboration Rule matches the Elaboration relation and transforms it to a panel containing a panel for the nucleus branch and a optional element for the satellite branch. The optional element is used to model a panel, which is only displayed after a condition is fulfilled. The panel for the nucleus branch is layouted to the left and the optional element is layouted to the right.

Title Rule: The Title Rule, illustrated in Figure 3.12, matches the Title relation and transforms it to a panel containing a panel for the satellite branch and a panel for the nucleus branch. The panel for the satellite branch is layouted on top of the panel of the nucleus branch because the satellite branch conveys title information related to the nucleus branch and thus is of major importance.

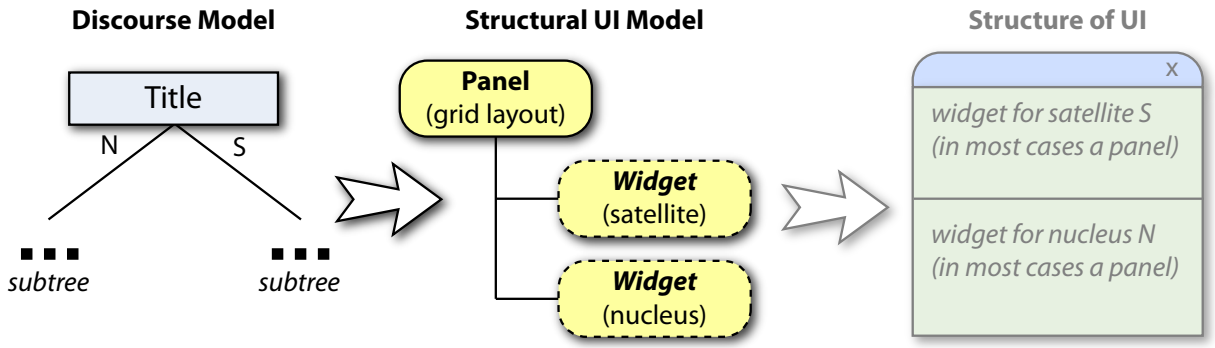


Figure 3.12: Title Rule

Annotation Rule: The Annotation Rule, illustrated in Figure 3.13, matches the Annotation relation and transforms it to a panel containing a panel for the satellite branch and a panel for the nucleus branch. The panel for the satellite branch is layouted below the panel of the nucleus branch because the satellite branch conveys annotation information related to the nucleus branch and this is of minor importance.

Result Rule: The Result Rule matches the Result relation and transforms it to a choice containing a panel for the nucleus branch and a panel for the satellite branch.

3.5.2 Content Transformation Rules Specific to Intentions

During the UI generation we had to deal with the presentation of content of the domain of discourse. In particular, the concrete presentation needs to depend on the purpose of the envisaged

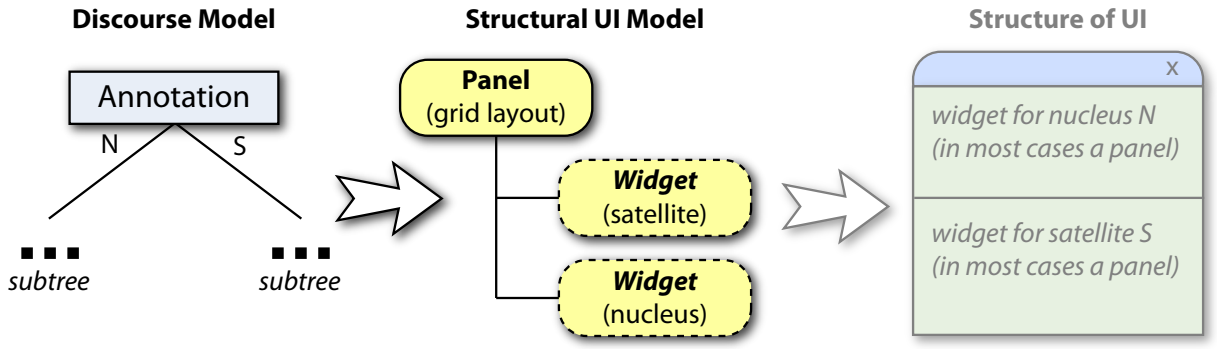
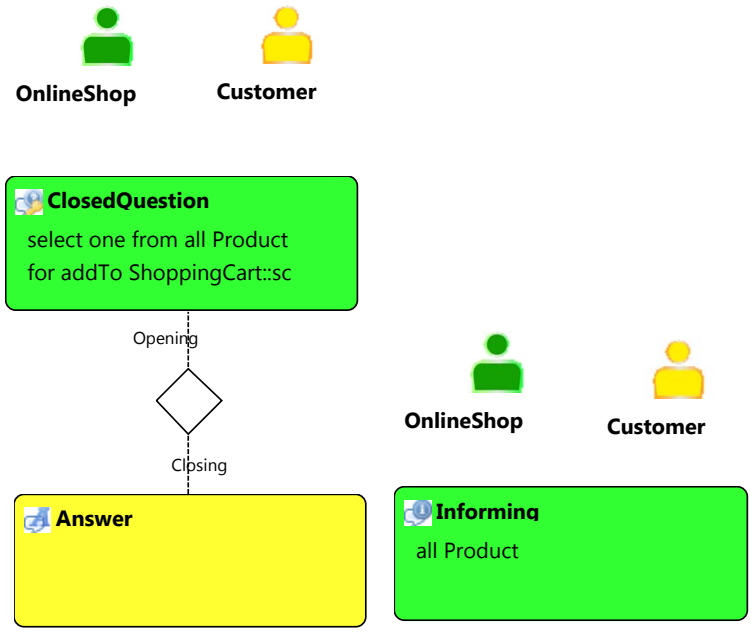


Figure 3.13: Annotation Rule

interaction, since the GUI's usability would clearly not be satisfactory otherwise, and usability is one of the essential problems of automatically generated user interfaces. In this subsection, the approach to automatic content generation with content transformation rules specific to intentions [KFK09] is presented.

Figure 3.14 shows an example in two small excerpts of a larger Discourse Model for a simple online shop. Figure 3.14a shows two Communicative Acts (represented by rounded boxes), a closed question and an answer, for adding a product to the customer's shopping cart. For this purpose, the formal expression within the closed question enables the customer to *select one* instance *from all* instances of the class *Product*, and provides the intention of the question, *adding* the selected instance to the *ShoppingCart* represented by the variable *sc*. Figure 3.14b shows a different part of the Discourse Model, for informing the customer on all products available in the store.



(a) Closed question about a product to select for adding to the shopping cart

(b) Informing on all products available in the store

Figure 3.14: Excerpts of an online shop Discourse Model

Communicative Acts typically refer to propositional content. In this example, it is about selecting a product by the customer and providing information on all products. In Figure 3.14 the propositional content is specified by the text below the type of each communicative act. In fact, it is the *same* propositional content for both communicative acts (*all Product*[6]) that gets uttered.

Propositional content is specified in our approach in a model of the domain of discourse, which specifies what the dialogues can "talk" about. Figure 3.15 shows a very small excerpt of such a model in a UML class diagram.[7] It specifies a single class named *Product* with four attributes.

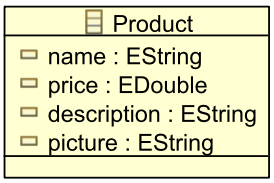


Figure 3.15: Small excerpt of a domain of discourse model

This example does not show a Discourse Relation. Both excerpts in Figure 3.14 originate from different parts of a larger Discourse Model and are, therefore, more indirectly connected with each other. Still, one can imagine to link the *Closed Question-Answer* Adjacency Pair in Figure 3.14a via a *Background* relation with an *Informing* on the category the customer can choose from to support her in selecting a product.

For transforming the Discourse Model excerpts in Figure 3.14a and 3.14b, we need structural transformation rules for transforming the *Question–Answer* Adjacency Pair and the *Informing* Communicative Act (that makes up a degraded Adjacency Pair by itself). Second, we need content transformation rules for transforming content types, like strings, pictures and numbers depending on the Communicative Act they are embedded in. The transformation rules also contain heuristics to improve the generated Structural UI Model. For example, rules can transform content attributes differently based on their attribute name, e.g. a *name* attribute can be used as a heading for the rendered content. The two structural rules below are applied in the first step to our Discourse Model excerpts in Figure 3.14:

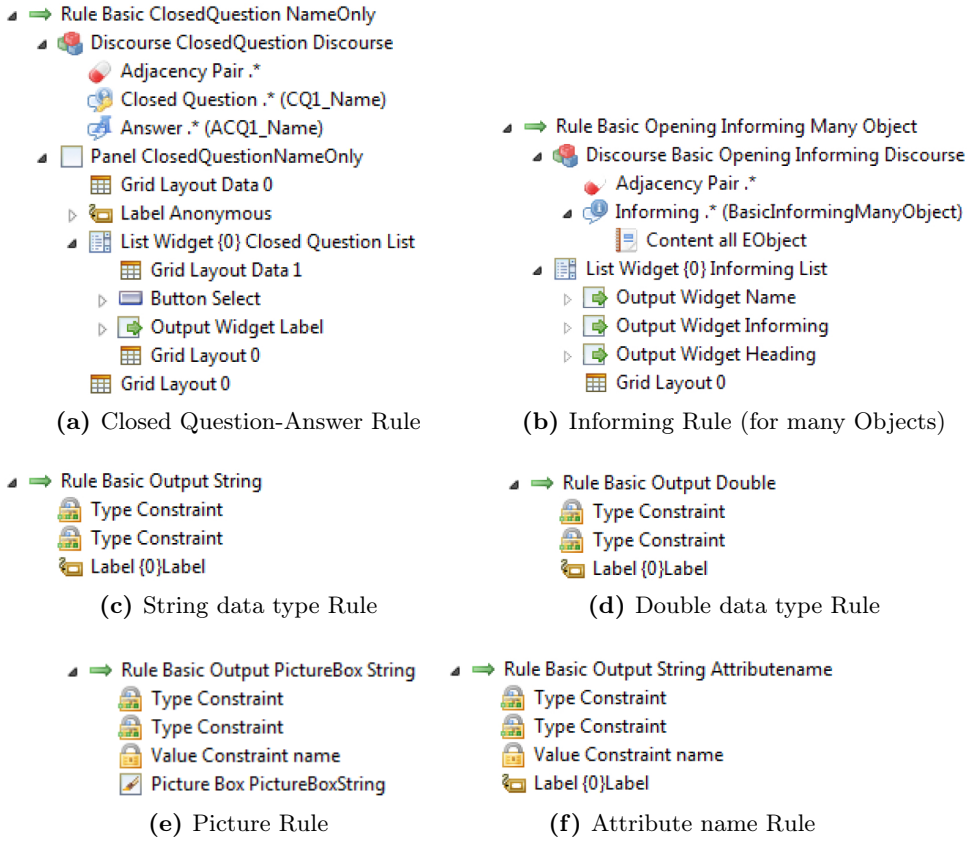
Closed Question-Answer Rule: The rule in Figure 3.16a transforms each *Closed Question-Answer* Adjacency Pair to a *Panel ClosedQuestionNameOnly* with a *Label Anonymous* and a *List Widget Closed Question List*, with each list entry consisting of an *Output Widget Label* placeholder for the content object's identifier (e.g., *name*) and a *Button Select* to select this list item. The output widget placeholder has a property that holds an OCL[8] expression, which selects parts of the content object the output widget is a placeholder for. For example, the *Output Widget Label* selects the *name* attribute of the content object, e.g., the product name, which is later on used for generating the label widgets. The corresponding OCL expression is "eAllAttributes→select(name='name')".[9] This and the other properties of the *Output Widget Label* are not shown in Figure 3.16a. The *Label Anonymous* represents a heading for the overall list and its text is derived from the name of the matched closed question Communicative Act.

[6] *Product* refers to the class with this name in the model of the domain of discourse.

[7] At the time of this writing, the specification of UML is available at `http://www.omg.org`.

[8] OCL — Object Constraint Language, see `http://www.omg.org/cgi-bin/doc?ptc/2003-10-14` for its specification.

[9] "eAllAttributes" is part of EClass in the underlying ECore meta-meta-model, which is an Essential Meta-Object Facility (EMOF) implementation in the Eclipse Modeling Framework (EMF).



(a) Closed Question-Answer Rule **(b)** Informing Rule (for many Objects)

(c) String data type Rule **(d)** Double data type Rule

(e) Picture Rule **(f)** Attribute name Rule

Figure 3.16: Transformation rules for closed question-answer (a) and informing (b), rules for transforming output widgets depending on data types (string (c) and double (d)), and for transformations based on heuristics (pictures (e) and attribute names (f))

The generated *Grid Layout* elements allow the specification of the desired number of columns for the resulting list and the placement of the label and the button next to each other as shown in the Final UI in Figure 3.18.

Informing Rule (for many Objects): The rule in Figure 3.16b matches an Adjacency Pair linked to an Informing (upper part of the figure). The Informing Communicative Act has to contain more than one content object (indicated by an *all* or *many* quantifier) for generating a list in the structural UI model. The rule transforms the matched input to a *List Widget* which contains the structure of one list item. The list widget contains *Output Widget placeholders* for the transformed content. The *Output Widget Heading* selects again the *name* attribute of the content object as representative for a list element's heading. The OCL expression for filtering out the *name* attribute is identical to the one in the Closed question–answer transformation rule. Each of the other two output widget placeholders is a placeholder for all the other attributes of the content object. In the second stage, both are rendered differently, the *Output Widget Name* is used to render the attributes' names and the *Output Widget Informing* is used to render the attributes' values. This distinct rendering can be seen in the Final UI in Figure 3.19 below with the attribute names on the left side and the attribute values on the right side.

Within the context of the rules described above, the following four rules are used in the online shop example excerpts in Figure 3.14 in the second step. They transform content object parts—mainly attributes—based on their content type and the type of Communicative Act that the content

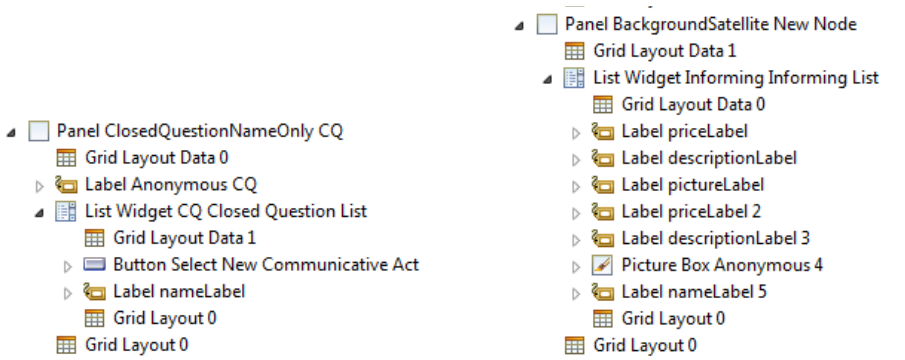
object is referred from.

String content type Rule: The rule in Figure 3.16c matches the *String* content type. The name of the rule reflects the matched content type. It generates a label for each string with the content value as the label's text. The context of this rule is specified by two *Type Constraints*. The first one specifies that this rule can be applied only to *Output Widget* placeholders generated in the first step, and the second one specifies that this rule can only be executed in the context of *Informing* and *Closed Question* Communicative Acts. In our example, this rule generates a label widget in the resulting structural UI model for each product attribute of type string that is associated with an output widget placeholder.

Double content type Rule: The rule in Figure 3.16d is identical to the *String content type Rule* apart from matching attributes of type *double* instead of type string. Thus, it creates a label for each numerical attribute. The name of the rule again reflects the matched content type.

Picture Rule: The rule in Figure 3.16e matches also content of type string like the *String content type Rule* but contains an additional constraint. This rule is only applied in the context of output widget placeholders and Informing Communicative Acts, which is specified by the two *Type Constraints* in Figure 3.16e. In addition, this rule contains a *Value Constraint* that checks if the name of the content attribute contains either the text "picture" or "image". If this constraint holds, the "picture" attribute's value is interpreted as a filename or URL and a *PictureBox* is generated in the resulting structural UI model for this content attribute. Since this rule is more specific than the string content type transformation rule, this rule is applied first when both match.

Attribute name Rule: The rule in Figure 3.16f matches any content attribute and is executed in the context of output widget placeholders and any kind of communicative acts. In contrast to the *String content type Rule*, this rule generates a *label* using the attribute's name instead of its value as label text. The attribute's name is retrieved in the rule by assigning the OCL expression "self.name" to the *Label* widget contained in the rule.[10]



(a) Structural UI Model corresponding to the Closed Question-Answer

(b) Structural UI Model corresponding to the Informing

Figure 3.17: Structural UI Models corresponding to the online shop Discourse Model excerpts in Figure 3.14

When these rules are applied to the example discourse excerpts in Figure 3.14, we get the generated Structural UI Model illustrated in Figure 3.17a for the Discourse Model excerpt in Figure 3.14a,

[10]The context for evaluating the OCL expression is the matched EAttribute object of the Ecore model.

and the Structural UI Model in Figure 3.17b for the Discourse Model excerpt in Figure 3.14b, respectively. The resulting structure of the *Closed Question-Answer* Adjacency Pair in Figure 3.17a corresponds to the structure shown in Figure 3.16a with the output widget placeholder replaced by the application of the *String content type Rule*. The resulting structure of the *Informing* in Figure 3.17b corresponds to the structure of the Informing rule in Figure 3.16b with the output widget placeholders replaced by applying all four content transformation rules to the attributes of an online shop product. The set of content transformation rules that can be applied is defined by the context formed by the superior structural transformation rule and the matched communicative act type (purpose of conveyed content). The context has to match all *type constraints* of the content transformation rules.

The rationale for generating the content presentation differently in this way is as follows (the screen shots in Figures 3.18 and 3.19 below may illustrate it better than the Structural UI Model in Figure 3.17). We render the content item in the context of the *Closed Question* less completely than in the context of the *Informing*, to provide a better overview for the customer during her product selection process. More precisely, in the context of the closed question only the name of the product is displayed, together with a select button. In contrast, the same content item is presented in the context of Informing by showing all its available information, even including a picture and the names of the attribute fields. So, the different context matters, since the purposes are different: asking vs. informing.

In case of content transformation rules which match content types, constraints can be specified on the rendering context, e.g., they can be used to constrain the set of types of Communicative Acts the content must appear in. This permits restricting a rule to a specific set of Communicative Acts. The four content transformation rules all match content of Informing Communicative Acts. However, only the *string and double content type transformation rules* can also be applied to content of closed questions.

When multiple rules trigger, a conflict resolution strategy is used to select one rule for firing. The conflict resolution strategy selects the most specific rule based on the size of the pattern to match and the number of constraints. In addition, a rule priority can be used to select one of many rules that have the same specialization degree. For this example, rule priorities are not necessary because the rules can be uniquely selected for firing without them.

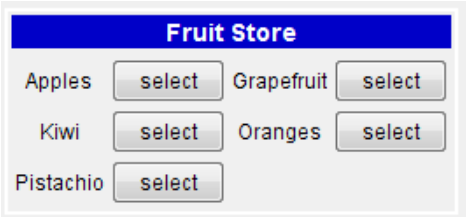


Figure 3.18: Screenshot of the Final UI representing the *ClosedQuestion*

When a rule fires, its widget tree structure is added to the resulting Structural UI Model and the widgets' content is selected from the Discourse Model by evaluating OCL expressions on the currently processed Discourse Model element. Using this rule-based transformation engine with the rules described leads to the device-specific but GUI toolkit-independent Structural UI Model as shown, for example, in Figure 3.17.

A Structural UI Model resulting from our model transformation process, like the one in Figure 3.17a or 3.17b, contains already the complete structure and layout information of the GUI

but is still GUI toolkit-independent.

Figure 3.18 displays the screen resulting from the Structural UI Model in Figure 3.17a by applying the screen generation for Java Swing. Figure 3.19 shows the screen resulting in analogous manner from the Structural UI Model in Figure 3.17b. In this example, we achieved a one-to-one mapping between structural model widgets and Java Swing widgets, only the PictureBox widget in Figure 3.17b is mapped to a Java ImageIcon embedded into a JLabel. In some cases, the mapping process is more complex, e.g., our structural UI metamodel contains an image map widget which requires an image, a label widget and the implementation of multiple active areas within Java Swing.

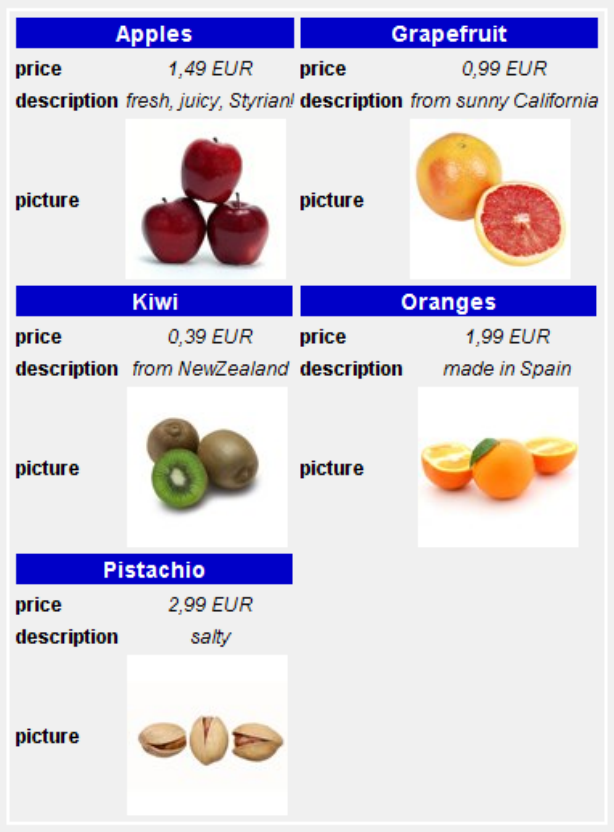


Figure 3.19: Screenshot of the Final UI representing the *Informing* on all products

3.5.3 Finger-based Touchscreen Specific Transformation Rules

This subsection explains how different transformation rules are used for automated GUI generation in a way that takes pointing granularity into consideration [KRF+09]. More precisely, WIMP (window, icon, menu, pointer) UIs are generated. The model-transformation rules can be divided into different rule sets according to specific device properties. One rule set is suitable for fine granularity, another rule set for coarse granularity. Both rule sets have a common core, however. These different rules can result in different layout, widget size and even widget type selection. The generator program chooses the respective rule set according to the specified properties. Example applications for the rules that take coarse granularity into account are finger-based touchscreen applications and applications for motor-impaired users.

A small excerpt of a larger Discourse Model for interacting with a robot shopping cart is shown in Figure 3.20. This Discourse Model is used to illustrate transformation rules specific to pointing granularity. The part shown in Figure 3.20 models presenting the current state of the customer's shopping list that she has entered before and the possibility of removing items from the shopping list.

In the Discourse Model excerpt example in Figure 3.20, the robot cart may ask a *closed question* or *inform* the customer, while the customer can provide an *answer* to the question. The association of a Communicative Act with a communication party is done by color. In this example, the green (or dark gray) Communicative Acts are uttered by the robot cart and the yellow (or light gray) ones are uttered by the customer.

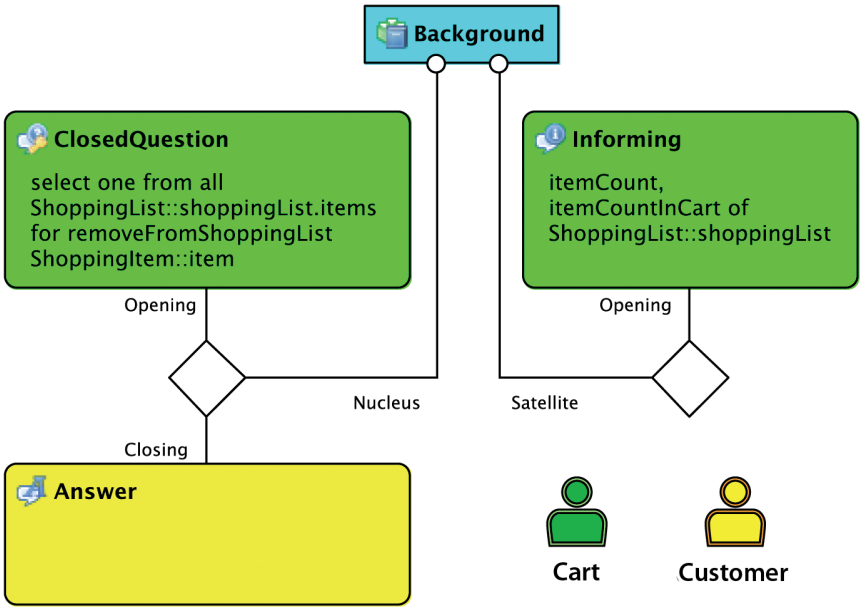


Figure 3.20: Discourse Model excerpt

The expression "`select one from all ShoppingList::shoppingList.items for removeFromShoppingList Shopping::item`" tells that the customer may select one item from the `items` of her `shoppingList` to remove it from the list; `shoppingList` is a variable of type `ShoppingList`, which is a reference to the *ShoppingList* class in the domain of discourse model.

In the Discourse Model excerpt example in Figure 3.20, the question–answer pair and the informing are related by a *Background* relation. This *Background* relation states that the Informing in the satellite branch contains background information to the "nuclear" utterances to support the customer in answering the closed question. It is transformed by a specific rule for the Background relation that renders the satellite branch below the nucleus branch instead of placing it next to it, due to a given GUI design that required so. This relation does not imply a temporal order per se. For instance, both pieces of information can be presented in parallel as done in the graphical user interface shown in Figure 3.24. On the other hand, if the cart uses speech for asking the customer, a natural order would be to ask the question first, then to provide the background information, and finally to wait for the answer of the customer. Thus, the Discourse Models specify classes of dialogues, with different possible orders of communicative act utterances.

In order to parameterize the semi-automatic user interface generation approach for diverse devices, some formal specification of devices and their properties is needed. Specifying physical devices, like

desktop PCs and PDAs, and their physical properties, e.g., screen size, resolution and supported GUI toolkits, allows us to generate graphical user interfaces with an appropriate screen layout, for example. A physical device specification is often insufficient for semi-automatic user interface generation, however, since it does not specify how an application makes use of the physical device. For instance, if a touchscreen is available, an application can use the device in different ways by imposing different interaction styles—pen-based interaction or finger-based interaction. These different ways of use shall be reflected in the user interface generation too, since finger-based interaction requires larger input widgets than pen-based interaction. Consequently, application-tailored device specifications in addition to physical device specifications are introduced.

Figure 3.21 illustrates both kinds of device specifications and their relation. The application-tailored device is derived from the physical device as a specialization and inherits the physical device properties from it. For instance, such properties are screen resolution and DPI, which together specify the metric screen size as well. The additional properties of the application-tailored device specify how the physical device is used by the application. For example, a physical device "touch screen" that can be either used with a pen or with fingers requires the additional property "pointing granularity" to be included in the application-tailored device specification. This property can have the value "coarse" for finger-based interaction, whereas for pen-based applications the "pointing granularity" will have the value "fine".

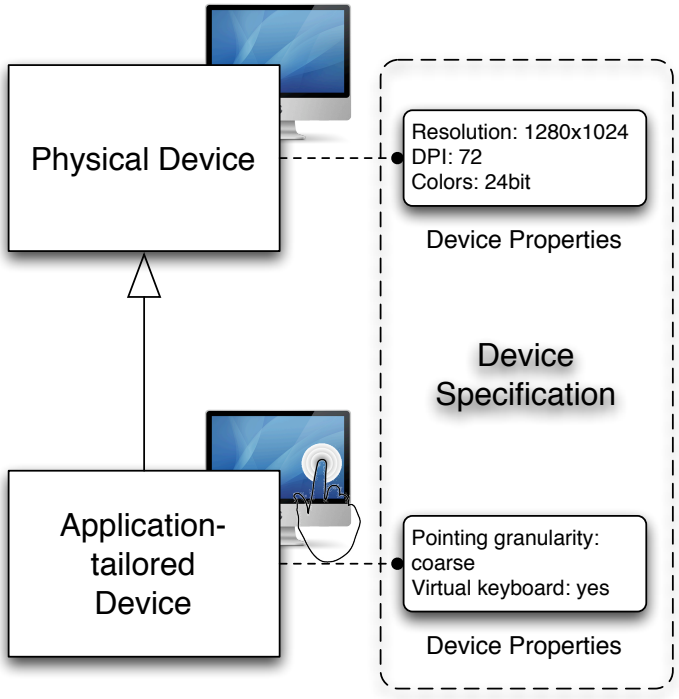


Figure 3.21: Physical and application-tailored device specifications

An application can further combine physical and virtual devices in a special way. For instance, a touchscreen solution usually does not include a physical keyboard. Nevertheless, an application could require the presence of a virtual keyboard, which in turn imposes constraints (especially spatial layouting constraints) on the automatic user interface generation. Therefore, the specification of application-tailored devices includes also properties defining the inclusion of virtual

devices.

Having such application-tailored device specifications available, user interface transformation rules and code generation can be restricted to a set or range of device property values. The model-transformation rules are divided into different rule sets accordingly. Applying rules from these different rule sets can result in different layout, widget size and even widget type selection. The automated UI generation chooses the respective rule set according to the specified device properties. For example, rules for a coarse pointing granularity—due to finger-based use of the device—can lead to larger widgets or avoidance of complex widgets like drop-down boxes.

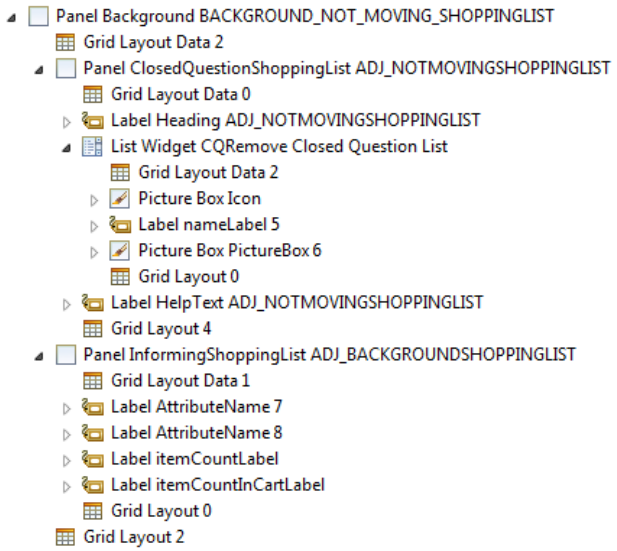


Figure 3.22: Structural UI Model excerpt automatically generated for touchscreen

For transforming the Discourse Model excerpt in Figure 3.20, structural transformation rules for transforming the *Closed Question-Answer* Adjacency Pair and the *Informing* Communicative Act as well as the *Background* relation are needed. Second, content transformation rules for transforming content types, like string, picture and number dependent on the Communicative Act they are embedded in are needed. The transformation rules also contain heuristics to improve the generated Structural UI Model. For example, rules can transform content attributes differently based on their attribute name, e.g., a *name* attribute can be used as the heading for the rendered content. The structural rules below illustrate the difference between rules applied to the same Communicative Act for different device specifications.

Closed Question-Answer Rule for coarse granularity: This rule transforms each *Closed Question-Answer* Adjacency Pair to a panel with a label and a list with each list entry consisting of an output widget placeholder for the content object's identifier (e.g., name). The label represents the heading for the overall list and the label's text is derived from the name of the closed question communicative act. A property is set in the list widget element of the structural UI model that enables the generation of Scroll Buttons for the list. The result of this rule is the rendered "Shopping List" in the left part of Figure 3.23. Due to the coarse pointing granularity, the clickable area for each list entry is extended to the size of the whole list entry.[11]

[11]The size of this figure as given here is on purpose nearly the real size on the physical device. The minimum target size on the device is as suggested in the literature referenced above.

Figure 3.23: Final finger-based touchscreen user interface

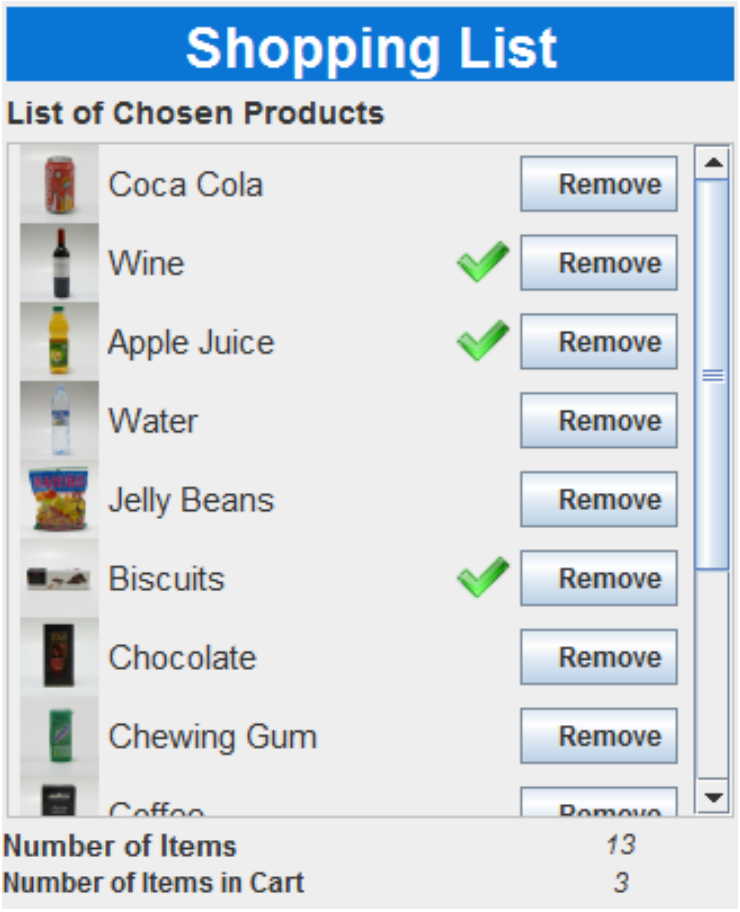


Figure 3.24: Excerpt of final desktop user interface

Closed Question-Answer Rule for fine granularity (scrollable list): This rule transforms each *Closed Question-Answer* Adjacency Pair to a panel with a label and a list with each list entry consisting of an output widget placeholder for the content object's identifier (e.g., name) and a button. In contrast to the "Closed Question-Answer Rule" in Section 3.5.2 this rule creates a scrollable list. The label represents the heading for the overall list and the label's text is derived from the name of the closed question Communicative Act. The result of applying this rule can be seen in the upper part of Figure 3.24. Due to the fine pointing granularity, the clickable area for each list entry is restricted to the size of the remove button. Additionally, a scrollbar is added to the right of the remove buttons.

Informing Rule (for one Object): This rule matches an adjacency pair with an Informing communicative act (upper right part of Figure 3.20). The rule transforms the matched input to a *Panel* containing two *Output Widget* placeholders. One is used to render the name and one the value of the attributes of the referred content. The output widget placeholders have a property that holds an OCL[12] expression which selects all attributes of the referred content. As can be seen at the bottom left part of Figure 3.23 and the bottom of Figure 3.24, the Informing is rendered equally in both cases, independently of the pointing granularity. This implies that output widgets

[12]OCL – Object Constraint Language, see `http://www.omg.org/cgi-bin/doc?ptc/2003-10-14` for its specification

are not influenced by the pointing granularity. Therefore, this rule belongs to the set of common core rules.

When these rules are applied to our Discourse Model excerpt in Figure 3.20, we get the generated Structural UI Model for touchscreens illustrated in Figure 3.22. The resulting structure of the *Closed Question-Answer* Adjacency Pair in Figure 3.22—the panel *ClosedQuestionShoppingList* and its children—corresponds to the structure described in the "Closed Question-Answer Rule for coarse granularity" and is illustrated in the Final UI as the Shopping List in Figure 3.23. The resulting structure of the Informing in Figure 3.22—the panel *InformingShoppingList* and its children—corresponds to the structure described in the "Informing Rule (for one Object)". It is shown in the Final UI by two labels below the Shopping List in Figure 3.23. The label with the text "Press List item to REMOVE Product" results from the "Closed Question-Answer Rule for coarse granularity". It is contained in the Structural UI Model in Figure 3.22 as the label "HelpText". The output widget placeholders are replaced by the appropriate second level rules.

The Structural UI Model generated for fine granularity is similar to the one for touchscreens with coarse granularity. It additionally has a *Remove* button contained in the list widget element of the closed question panel. The resulting buttons in the Final UI are shown on the right in Figure 3.24.

3.5.4 Transformation Rules for Different Screen Sizes

Automated generation of UIs has certainly advanced in recent years, especially based on model-driven approaches. Still, such generated UIs pose many usability problems. This is partly due to insufficient flexibility of the current generation approaches.

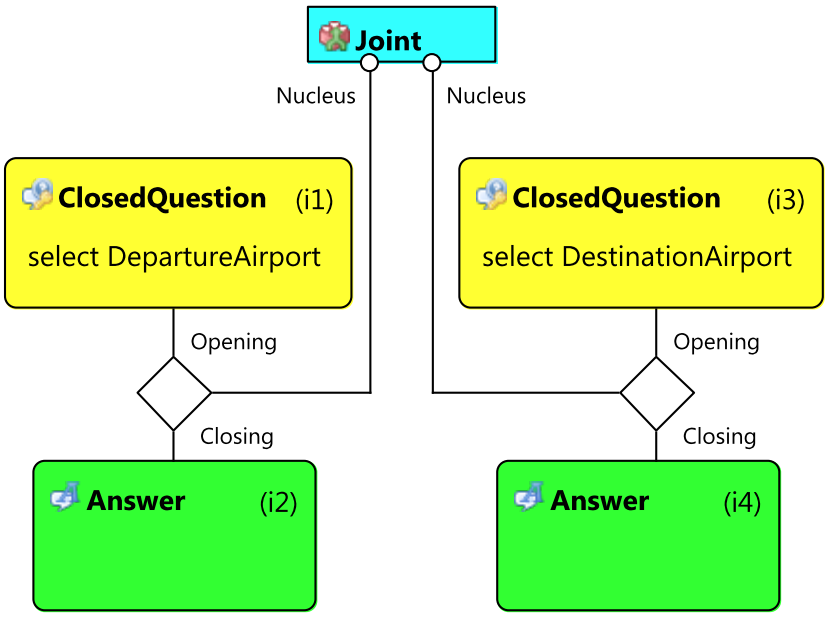


Figure 3.25: A Discourse Model excerpt

In particular, straight-forward model-driven generation only allows for matching a single transformation rule for each source pattern. This approach is extended by taking up means from rule-based programming, that have been around for a long time. Matching of several transformation rules for any source pattern is allowed, and so-called conflict resolution is used to determine

which rule to apply (fire). Based on that, a simple form of optimization is implemented in the context of model-driven UI generation.

It allows to maximize the amount of information to be displayed on a screen with limited resolution, to minimize the number of navigation clicks, and to minimize scrolling. All this is important for reducing usability problems. Since more and more devices with different screen resolutions are used to run the same application, the generated UI can automatically be optimized for the given (limited) resolution.

A small excerpt of a larger Discourse Model for flight booking is shown in Figure 3.25. This Discourse Model is used as an example to illustrate transformation rules for different screen sizes, presented in [KRP+10].

In this example, the application asks the customer *closed question*s, while the customer provides *answer*s to the questions. In Figure 3.25, two Question–Answer pairs are related by a *Joint* relation. This *Joint* relation states that the Question–Answer pairs in both nucleus branches are of equal importance. Further, it does not imply a temporal order per se. For instance, both pieces of information can be presented in parallel if there is enough space on the screen. Otherwise they can be uttered in sequence.

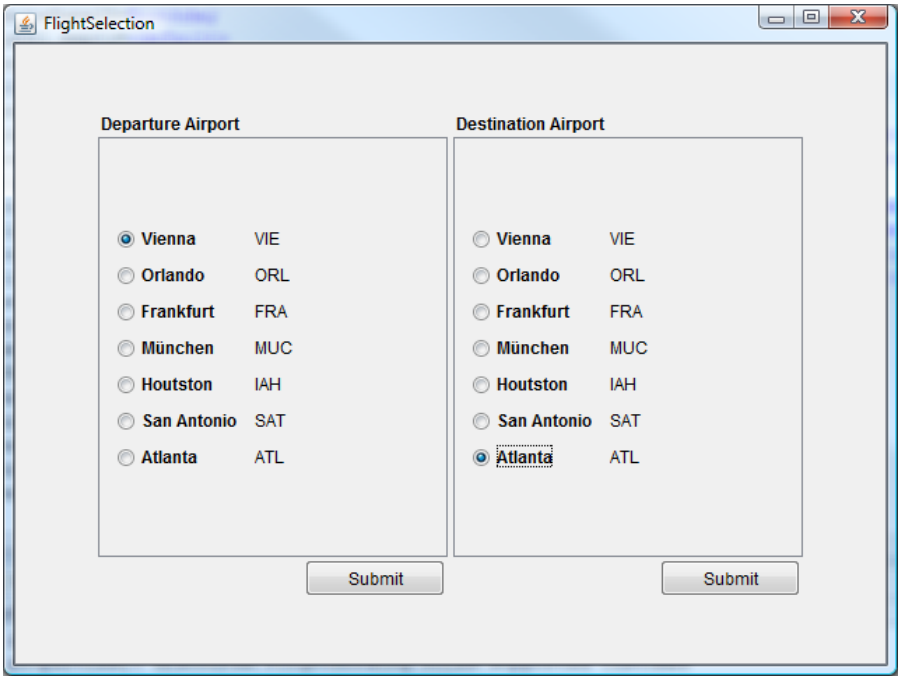


Figure 3.26: Generated UI for 640×480

The following transformation rules are applied to elements of the discourse model excerpt in Figure 3.25 for generating a model representing the structure of the Final UI in Figure 3.26.

First, the *Joint Rule* gets applied that matches the Joint relation and adds a panel to the Structural UI Model. This panel acts as a container for the Radio Button lists in Figure 3.26, which correspond to the two nucleus branches of the Joint relation.

Second, a *Closed Question Rule (radio buttons)* gets applied twice that matches each of the two Question–Answer adjacency pairs. For each adjacency pair a panel containing a label for a heading, a list of radio buttons together with item labels, and a submit button on the bottom is

added to the Structural UI Model. The reason why two submit buttons are generated instead of one is based on the approach with transformation rules. It can only be solved by considering the behavior of the UI, which is out of scope of this doctoral dissertation.

In the second step this Structural UI Model is used to generate source code for a particular target platform, e.g., Java Swing in this running example.

The problem tackled by this extended approach is to fit a given amount of information optimally (in the following sense) into screens with limited resolution.

We assume that the following optimization objectives improve the usability of the generated user interfaces:

- maximum use of the available space for the given resolution,
- minimum amount of navigation clicks, and
- minimum scrolling (except list widgets).

Whenever the given information to be displayed does not fit into a single screen with default widgets, it is tried to display it with widgets that use less space. If it still does not fit into a single screen, then its display is split to two or more screens. Splitting increases the number of navigation clicks but it minimizes scrolling. List widgets are excluded from this last optimization objective because the number of list entries can vary extremely at runtime and determines whether the list is scrollable or not. This information is not known during the rendering process.

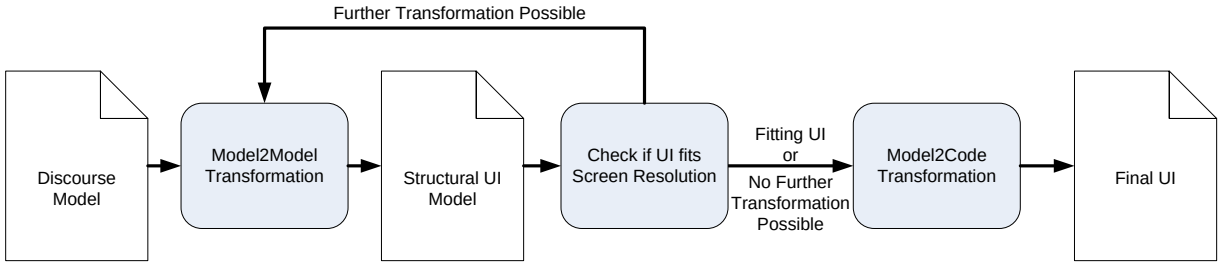


Figure 3.27: The Extended Transformation Process [RPK+11b]

The basic transformation process illustrated in Figure 3.1 in Section 3.1 looks like straight-forward model-driven generation that only allows for matching a single transformation rule for each source pattern. Therefore, the straight-forward approach is extended by allowing that several transformation rules may match for each source pattern, and by applying so-called conflict resolution to select which rule to apply (fire) in the next model-to-model transformation. The extended generation process shown in Figure 3.27 illustrates the resulting possibility of trying out several rules for optimization purposes. In this approach, the rules need not to be specifically designed for a particular screen resolution. It is rather the way the rules are applied that achieves the given optimization objectives.

In order to implement such an optimization, the conflict resolution mechanism needs to select the rules in a certain defined order. For achieving the optimization objectives given above, this selection order is according to the space that the widgets the rule creates occupy in the Final UI.

Therefore, all rules matching the same discourse element for transformation have to be ranked by the designer according to this space need.

Each target device that is rendered for has an abstract device specification that contains all style data used by the transformation rules. These data specify default sizes for all input and output widgets on the target device that can be overwritten in a transformation rule. They are used to set the size for each Final UI element and allow us to calculate the exact size of each container (e.g., panel). For example, we set the size of the list widget explicitly. This makes it independent from the number of entries. If the list widget is not able to display all entries, it becomes scrollable.

After the size calculation each generated screen is tried to layout to fit into the given resolution. However, only the arrangement of the widgets that has not been fixed explicitly in a transformation rule is modified. Therefore, the layout specified by the *Closed Question Rule (radio buttons)* (i.e., the layout of the heading label, the radio button list and the submit button in Figure 3.26) is not changed. In this example, the position of the complete radio button lists in the panel created by the Joint relation is modified, since the *Joint Rule* does not contain any layout information.

Now let us explain how to apply this approach to automatically generate user interfaces for three target resolutions. As input the Discourse Model excerpt shown in Figure 3.25 is used.

The first GUI is rendered for the resolution 640×480. The first cycle of the model-to-model transformation uses the highest ranked rules (i.e., the ones with the highest space need) for each discourse element. These are the same rules that have been applied in our basic transformation process. After the first transformation cycle the size for each panel in the corresponding Structural UI model is calculated. They can be placed next to each other without exceeding the screen resolution. So, this is a fitting UI and we trigger the model-to-code transformation. The result is shown in Figure 3.26.

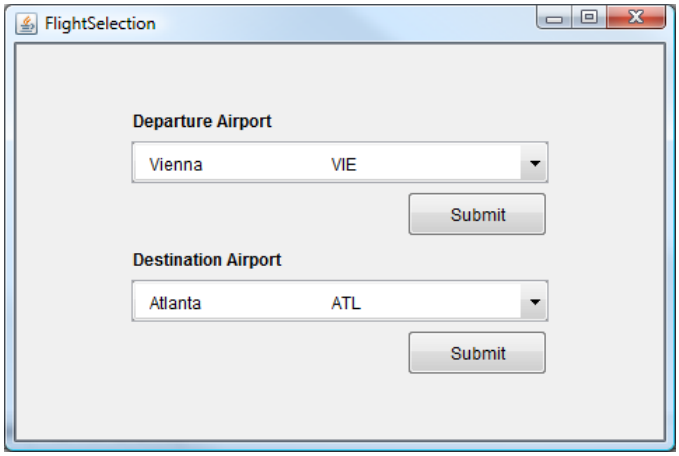


Figure 3.28: Generated UI for 480×320

Next a UI for the resolution 480×320 is generated. This time, the UI resulting from application of the highest ranked rules does not fit. As long as a lower ranked rule can be applied, we initiate another generation cycle. First, the *Small Closed Question Rule* is used in our example. This rule matches the same source element (Question–Answer adjacency pair) as the *Closed Question Rule (radio buttons)* but it creates a UI structure which occupies less space on the screen. A combo box element presents the content of the Closed Question communicative act to the user and a submit button is generated to confirm the selection of the user. The user interface shown in Figure 3.28

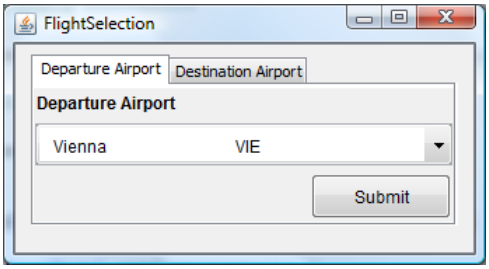


Figure 3.29: Generated UI for 320×180

is the result of two more cycles, because in each cycle only one lower ranked rule is applied. The resulting UI fits and still presents the same information, but using widgets with less space needs (combo boxes instead of radio buttons). However, the list widgets do not fit next to each other and the layouter arranges them vertically.

In a third run, a user interface for the resolution 320×180 is generated. Even after all rules for widget selection have been tried out, the generated GUI still does not fit the given resolution. Therefore, rules are used that split the screen in order to increase the number of navigation clicks before making use of scrolling. In this example, this means that in the next cycle the *Small Joint Rule* is applied instead of the *Joint Rule*. The *Small Joint Rule* matches the same source element (Joint relation) but creates a different UI structure (a tabbed pane element instead of a panel). Figure 3.29 shows the outcome for the resolution 320×180. The *Small Joint Rule* and the *Small Closed Question Rule* have been applied and a fitting UI has been generated after a third cycle of rule application. This time no layout modifications are necessary because each tab contains only one panel.

The worst case in the extended generation process occurs if and when no more rules are available and the generated screen still does not fit the given screen resolution. In this case, we stop the optimization loop and rely on scrolling.

A more sophisticated optimization strategy for small screens is presented in [RPK+11b].

3.6 Model-to-Code Transformation

The second step of the generation process is the actual screen generation. This process creates the target toolkit implementation of the Final UI, that represents all windows and frames needed to communicate with a user.

The code generator's task is the translation of a Structural UI Model to source code in a specified programming language. Since graphical user interfaces usually consist of a limited number of widget types combined in various ways, the resulting code can be seen as a combination of corresponding code fragments.

The most appropriate generation approach, regarding the input of the process, is the combination of meta-model and template-based generation. As large parts of our discourse modeling and user interface generation platform are based on the Eclipse Modeling Framework[13] and, therefore on Java, Java Emitter Templates (JET) have been chosen as the template language. The code generator was implemented as a template engine, using Java Swing[14] as target toolkit.

[13] `http://www.eclipse.org/modeling/emf/`
[14] `http://java.sun.com/products/jfc/`

The templates give the system designer a fair amount of flexibility as they support the separation between functionality and content. Static data already available at generation time can be retrieved directly from the Structural UI Model. On the contrary, list widgets, for example, are filled with dynamic data available only during runtime of the system. This allows the modification of data without needing to generate the system again.

The Structural UI Model takes layouting into account, but hardly contains anything concerning the *look* of the application. All this data can be encapsulated in a style sheet, whose elements are then associated with the Structural UI Model elements. Moreover, the look aspect is something that does not influence the logic of a system at all and can, therefore, be treated separately.

Cascading Style Sheets (CSS) encapsulate all information concerning the *look* of an object, making it easy to adapt a system to a new look by simply exchanging the style sheet. Since CSS have been designed for HTML pages, CSS attributes were mapped to Java Swing attributes within the generator templates. Apart from this mapping, the code generator provides default values for attributes like font type and size, as well as for border thickness and color. These default values are needed in case the system designer does not specify all attributes needed by the generator. If a style sheet is provided, all specified attributes are applied to the corresponding widget, overriding the default values.

The code generator further supports the fall-back mechanism known from HTML. In a first step, the generator tries to extract all the values from the CSS style that is associated with the *Structural UI Model widget's identifier*. If no widget-specific style is found, the code generator tries to extract the values from the *style class* associated with the widget. In case no style class is defined for the widget in the Structural UI Model, the code generator checks for a corresponding CSS *element style*. If all attempts fail, the value is set to the default value that is predefined in the code generator.

The code generator extends the basic functionality offered by CSS. An extension that influences the *look* of an application is the representation of enumeration values through icons (e.g., tick marks in Figures 3.23 and 3.24). Furthermore, decorative images or sounds can be specified for different widgets.

Several further CSS templates are provided that define the sizes of several widget types. In this way the style sheets capture not only characteristics regarding the *look*, but also the *feel* of an application, e.g., in case of a touchscreen UI. The code generator selects the appropriate style sheet template according to the application-tailored device specification and extracts the needed information at generation time, specifying the button size or the size of each list widget entry in Figure 3.23. The style sheets also define default values regarding the size and font of every widget type. In this way, the minimum size of buttons or other interactive widgets like the shopping and destination list in Figure 3.23 are set. As these settings are specified for a widget type, they are valid for all widgets in the Structural UI Model unless they are overridden. Therefore, they can be seen as default values for the application-tailored device. They can be refined either directly, by editing values in the style sheet, or by attaching a style reference to the Structural UI Model widget. In effect, the default values set by the template are substituted by the values defined in the corresponding style class. This allows the system designer to customize the standard values for selected widgets or widget classes.

The possibilities of style sheets however, do not cover all aspects concerning the *feel* of a user interface on different application-tailored devices as they do not allow modifying the widget hierarchy of the structural UI. This aspect is covered by the mapping rules applied during the first

generation step. An example are the additionally generated scroll-buttons for each list in Figure 3.23. Their generation results from the application of different transformation rules than the ones applied for generating the desktop UI illustrated in Figure 3.24.

The style sheets are applied only during the second step of the generation, i.e., the Structural UI Model to Java Swing translation. Any change to characteristics that are captured by the style sheet, requires only the repetition of the second step instead of the whole generation process.

Screen generation is our final step that transforms the Structural UI Model into GUI toolkit-specific windows and dialogs and generates code for them. Currently, the Java Swing[15] and Eclipse SWT[16] GUI toolkits are supported. This screen generation step solves four tasks:

- It maps the abstract widgets of the structural model to toolkit-specific widgets,
- it maps the generic structural UI layout to a toolkit-specific layout,
- it formats the toolkit-specific widgets according to cascading stylesheets (CSS), and
- it generates the event handling and the binding to the user interface behavior (represented as a generated finite-state machine that is derived from the Discourse Model obeying the procedural semantics of the RST relations as described in [PFA+09]).

For transforming the layout information of the Structural UI Model, an algorithm is implemented that calculates layout data required for the toolkit-specific layout managers. E.g., for the Java Swing *GridBagLayout* the weight of each grid cell is calculated, which is important for resizing windows, depending on the widget types and the layouting rule applied to the RST relation.

Further, the screen generation process formats the generated widgets according to style information stored in a cascading stylesheet (CSS). The appropriate style is selected according to the selection algorithm defined in the CSS specification based on the widget name, the style identifier and the widget type defined in the Structural UI Model. So, this provides a GUI toolkit-independent mechanism for specifying widget styles that is transformed by the screen generation into toolkit-specific method calls to set the toolkit widget's attributes.

In addition, the screen generation results in toolkit-specific event handlers that

- collect information from the input widgets,
- modify content objects provided by the application logic according to the collected information, or
- generate new content objects based on the collected information,
- and sends them to the application logic in response to a question or as a new request.

A detailed description of the model-to-code transformation for this approach to generate Java Swing code is presented in the diploma thesis [Ran08].

[15] `http://java.sun.com/products/jfc/`
[16] `http://www.eclipse.org/swt`

4 Requirements-based GUI Prototype Generation with Procedural Rules

Usually, model-driven generation of user interfaces (UIs) leads from higher-level task models to UI models (at certain levels of abstraction) and a Final UI. So, in addition to operationalizing as usual, we make use of artifacts in one "world" for creating related ones in another. Additionally, we allow transformations in either direction between the same pair of models.

For such model transformations in one direction, we had to implement transformation rules. For transforming back, rules for inverse transformations are needed as well. Rather than hand-crafting them, we propose to generate them semi-automatically. In fact, we applied model transformations to this task as well, through metarules.

We present this approach of bidirectional transformations between models of the same level of abstraction in the context of the previously defined Requirements Specification Language (RSL) [KSS+07]. In contrast to most other languages for requirements specification, RSL is a language that integrates requirements with UI specifications [MK08]. This integration along the representation dimension is supposed to facilitate combined work on requirements and user interfaces along the process dimension as well. In particular, we present an additional contribution to this doctoral dissertation, transformations between artifacts of requirements and UI specifications within RSL, presented in [KKMF09].

This chapter is organized in the following manner. First those parts of the RSL Metamodel needed within the scope of this doctoral dissertation are sketched. Then the overall approach of transformations in this context is presented. For elaborating on this overall approach, we specify (MOLA[1]) transformation rules from the requirements specification to an abstract UI specification, and vice versa. In addition, we specify such rules for transforming from an abstract UI specification to a more concrete UI prototype. Finally we present an approach to deriving inverse transformation rules semi-automatically using metarules.

4.1 The RSL Metamodel

In this section some background material about RSL is presented. In particular, those parts of its metamodel are explained for specifying requirements and UI upon which the transformations are built.

[1] `http://mola.mii.lu.lv/`

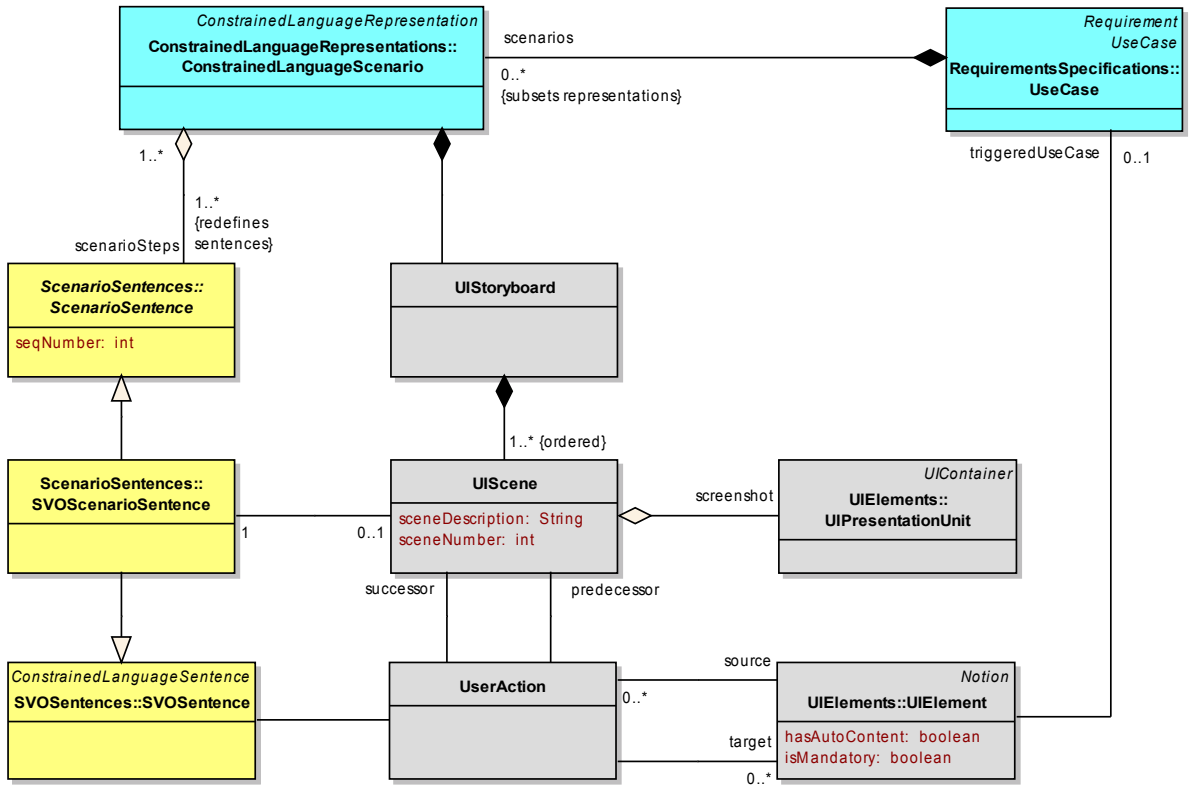


Figure 4.1: Part of the RSL Metamodel linking RE and UI Specifications [KSS+07]

Scenarios and use cases are popular in requirements engineering these days. The ReDSeeDS Requirements Specification Language (RSL) makes use of this popularity and includes ConstrainedLanguageScenarios contained in Use Cases. In the upper part of Figure 4.1, the composition relationship between ConstrainedLanguageScenario class and UseCase class in the RSL Metamodel is shown. ConstrainedLanguageScenarios provide a textual representation of use case scenarios illustrated in the left part of the RSL Metamodel in Figure 4.1. They consist of a sequence of SVO sentences describing the flow of interaction between the user and the system to be developed. An SVO sentence has a **S**ubject and a Predicate, which in turn has one **V**erb, and an **O**bject, hence SVO. Each word used in a sentence can be mapped to its specific meaning in the domain vocabulary. It is possible to define the meaning of each word in this context, also by reusing terminology from WordNet [WSBK08].

Moreover, RSL provides generic elements for specifying user interfaces (UI) that are both modality and toolkit independent. The static aspects of the UI, i.e., its structure and layout, can be described by using elements like InputUIElement for data input illustrated in the lower left part of Figure 4.2; TriggerUIElement for triggering actions illustrated in the lower middle of Figure 4.2; SelectionUIElement for exclusive or non-exclusive selection from more than two options illustrated in the lowest middle of Figure 4.2; and UIContainer or UIPresentationUnit illustrated in the lower right of Figure 4.2, which are used as containers of other UI Elements illustrated in the middle of Figure 4.2. A concrete example for an InputUIElement would be a "TextBox" Widget. A concrete example for a TriggerUIElement would be a "Button" Widget. A concrete example for a SelectionUIElement would be a "ComboBox" Widget. A concrete example for a UIContainer or UIPresentationUnit would be a "Panel" Widget.

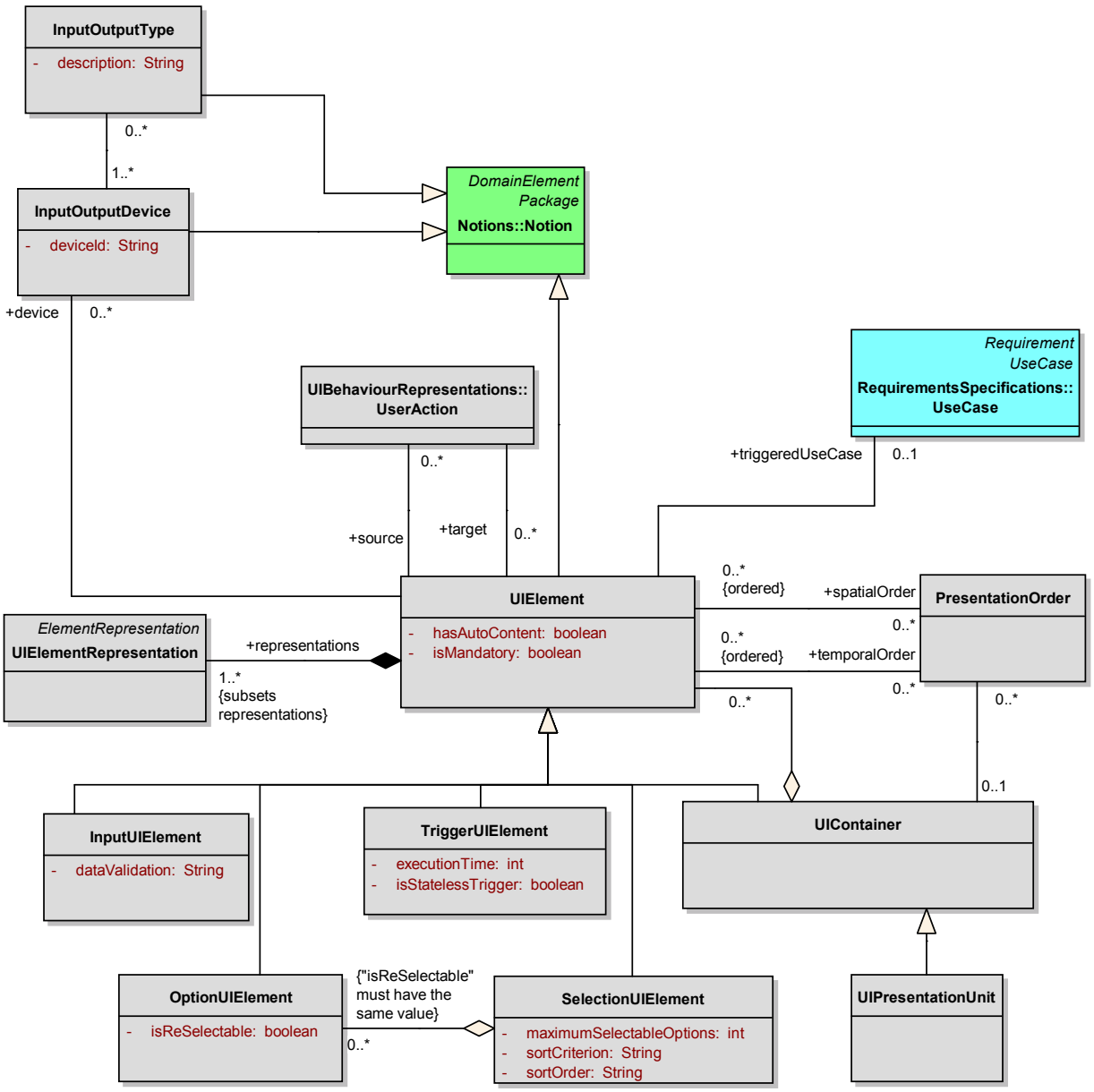


Figure 4.2: Part of the RSL Metamodel showing UIElements [KSS+07]

The dynamics of the UI, i.e., the behavior related to user interaction, can be described by using RSL elements like UIStoryboard, UIScene, and UserAction illustrated in the middle of Figure 4.1. A UIStoryboard is a series of scenes displayed in a sequence.

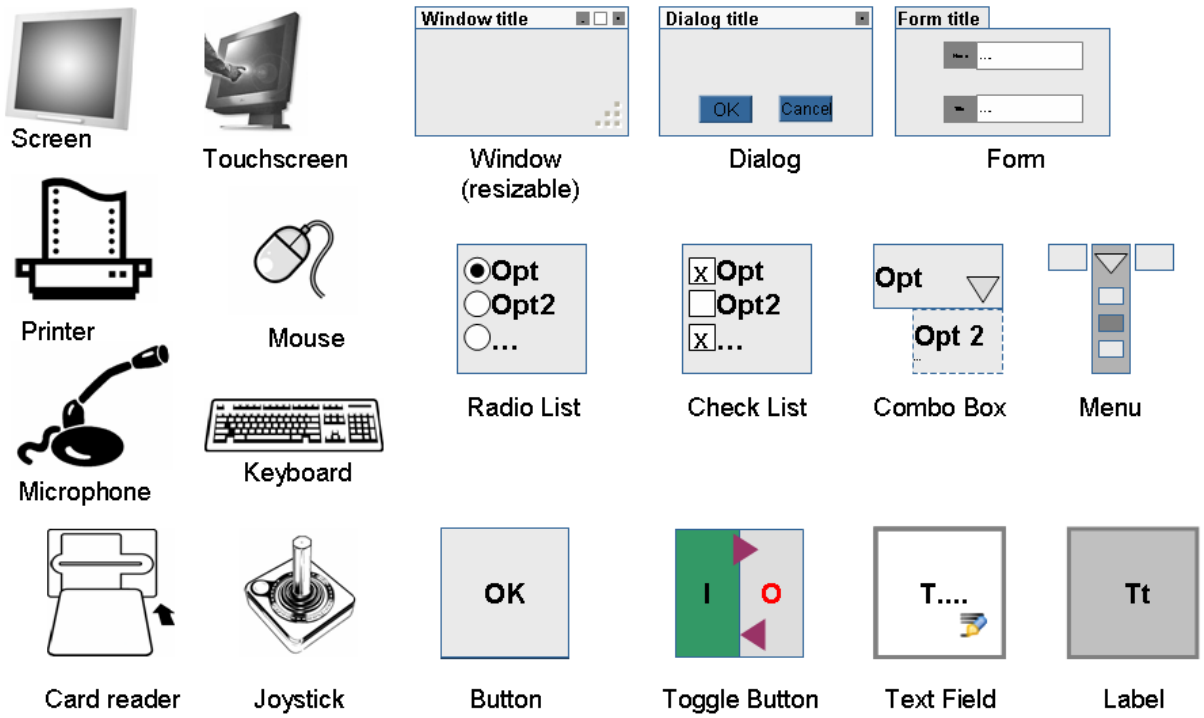


Figure 4.3: A selection of elements of the GUI Profile provided by RSL [KSS+07]

The "presentations" of the individual scenes are defined in UIScenes. UIScenes can be connected by UserActions indicating the triggering action of the user. A UserAction is performed on one source UIElement in a predecessor UIScene and can result in a transition to a successor UIScene as well as influence some of its UIElements. Of course, the source and target UIElements of a UserAction can be the same. Figure 4.1 shows the relationship between dynamic UIElements of a UI Specification and the ConstrainedLanguageScenarios of a Requirements Specification as one important link between the two "worlds".

Since these UI elements are modality independent, they specify an *abstract* UI (according to [CCT+03]), that can be used for the different modalities found in advanced UIs. In order to make a UI specification more understandable for a user, a modality specific *concrete* UI [CCT+03] is better suited. For this purpose, RSL also includes related elements, whose concrete syntax can even serve for specifying a UI Prototype. A selection of elements of the GUI Profile of RSL can be seen in Figure 4.3.

4.2 Transformations from Requirements Specifications via UI Specifications to UI Prototypes

Based on the RSL language and its metamodel, let me illustrate my overall approach of transforming between Requirements Specifications, UI Specifications and UI Prototypes. Figure 4.4

illustrates this approach.

Assume that a requirements engineer has specified a scenario using SVOScenarioSentences [KSS+07]. Such a scenario may contain SVO sentences like *User confirms selection* and *User selects exercise* (see Figure 4.4). Instead of having a UI designer manually create a related UI specification, transformation rules according to T1 in Figure 4.4 may be applied. They would lead to a partial model of an abstract UI. Applying transformation rules according to T2 would then lead to part of a more concrete UI, i.e., a UI Prototype. Note that this prototype is not the final user interface of the real application, but it serves the purpose of illustrating the textual scenario in the requirements specification to users and to capture the essence [CL99]. This difference must also be communicated to the user in order to avoid unnecessary discussions, for example on the Look and Feel of the UI.

Now assume the other way round. Users themselves or together with a UI designer have developed a prototypical storyboard, already using the GUI modality in its concrete syntax. This approach facilitates their intuition of how a system to be built may be used. For developing such a piece of software, however, a requirements specification may still be needed by (or simply useful for) the developers. Instead of having a requirements engineer manually write such a specification, transformation rules according to T2' in Figure 4.4 may be applied. They would lead to a partial model of an abstract UI, this time based on the UI Prototype. Applying transformation rules according to T1' would then lead to a part of the requirements specification, a scenario. Again, the generated scenario might not be in the final version yet, but it is consistent with the UI Prototype.

The transformation T3 can be seen as a composition of transformation rules for T1 and T2. The transformation T3' can be seen as a combination of the rules for T1' and T2'.

Now the transformation rules for the approach as visualized in the example of Figure 4.4 are described. The example consists of a ConstrainedLanguageScenario composed of the following two SVOScenarioSentences:

1. *User confirms selection.*
2. *User selects exercise.*

The following rules implement the transformation T1:

Rule 1: Each SVOScenarioSentence whose verb implies "selection" is transformed into a SelectionUIElement. Its name is the Noun in the VerbPhrase (see [KSS+07] for the definition of VerbPhrase).

Rule 2: Each SVOScenarioSentence whose verb implies "triggering an action/event" is transformed into a TriggerUIElement. Its name is the Noun in the VerbPhrase.

Rule 3: Each SVOScenarioSentence whose verb implies "showing information" is transformed into a UIPresentationUnit. The name of this UIPresentationUnit is the Noun of the Object in the VerbPhrase. Note, that these rules both require and utilize a mapping between synonyms. We reuse this mapping from WordNet.

Rule 4: Each ConstrainedLanguageScenario is transformed into a UIStoryboard of a similar name.

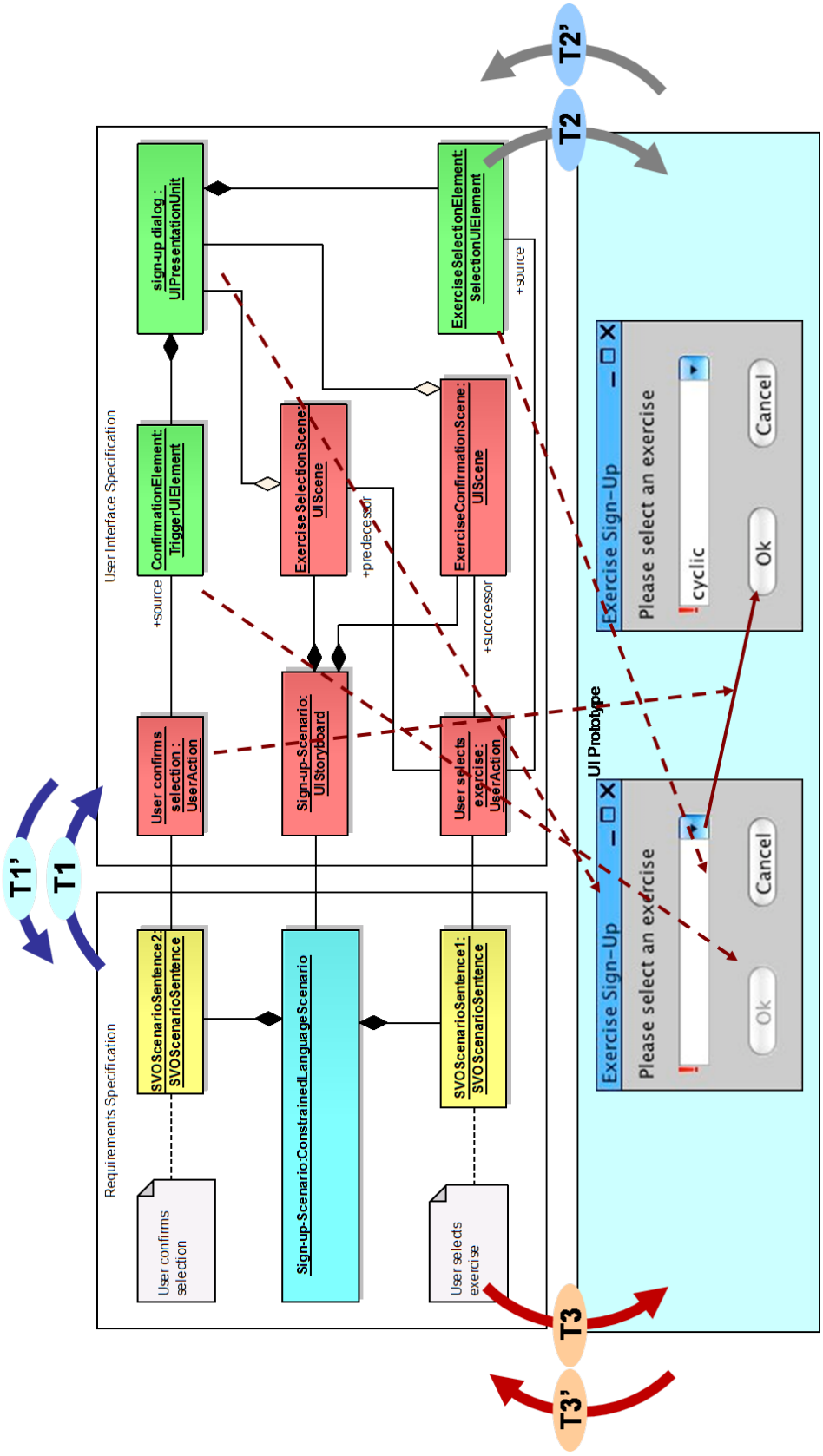


Figure 4.4: Overview of specifications and their transformations.

Rule 5: Each SVOScenarioSentence in a ConstrainedLanguageScenario is transformed into a UIScene. Its sceneDescription is the text of the Sentence and its sceneNumber equals the seqNumber of that sentence. The UIScene is linked to the UIPresentationUnit corresponding to this sentence.

Rule 6: Each SVOScenarioSentence whose subject is not "System" is transformed into a UserAction. Its source is the UIElement (other than the UIPresentationUnit) associated with the sentence. The predecessor of this UserAction is the UIScene of the previous sentence and its successor is the UIScene of the current sentence.

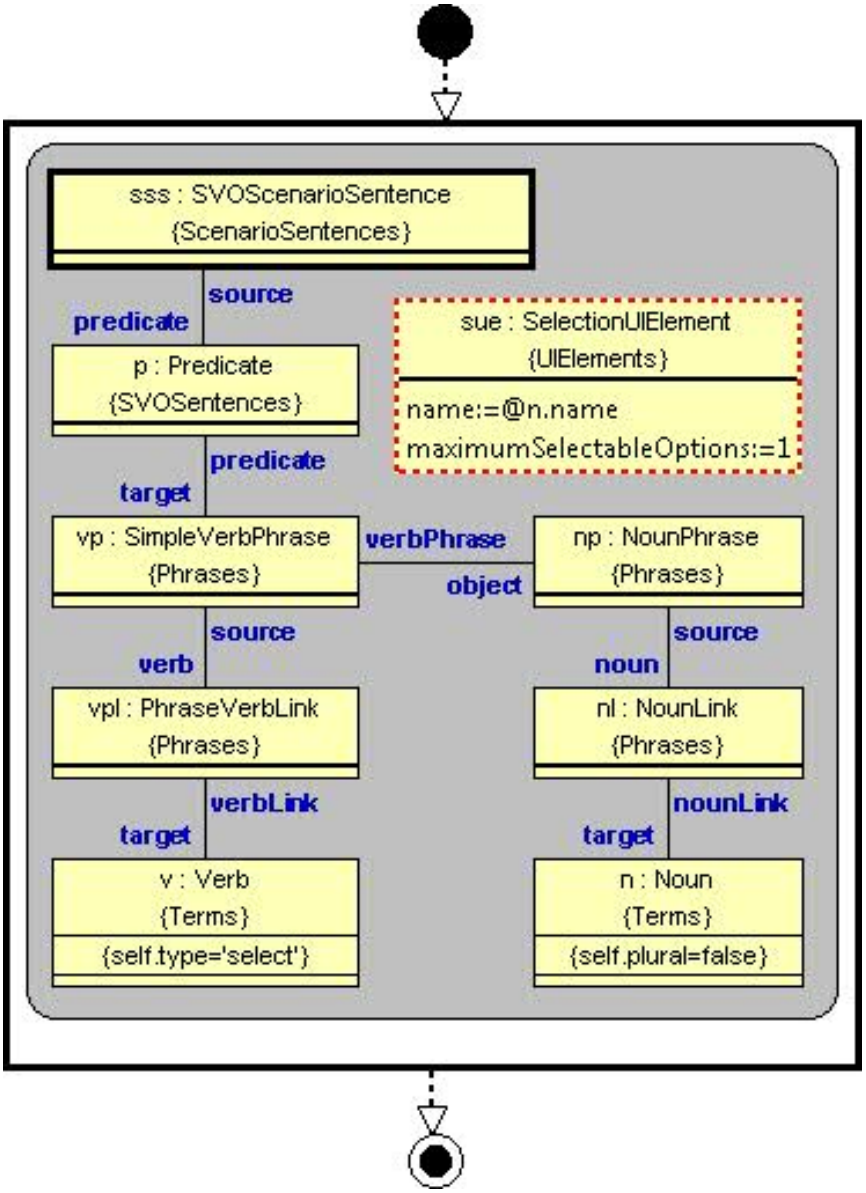


Figure 4.5: MOLA Rule R1 for T1.

Applying these rules to this example results in the UI Specification Model in the right box of Figure 4.4. The rule execution order has no impact on the generated model.

Figure 4.5 illustrates the formalized Rule 1 for T1 in MOLA, composed of the following elements. The outer bold rectangle symbolizes a *for-each* loop. The rounded rectangle inside represents

the actual rule, that will be repeated for each matched element. The small boxes inside the rule represent different kinds of classes, depending on their borderline style. When the thickness of the border line is regular, they represent a "normal" class. A bold border lined box represents a loop variable. A dashed lined box border represents a class that will be created by the transformation rule. The small black circle represents the starting point of the rule. The double rounded circle represents the end point of the rule. In particular, Rule 1 iterates over all SVOScenarioSentences. Whenever the type of the verb is "select" and the noun is not plural, the rule matches. In this case, a SelectionUIElement of the UI Specification Model is generated. The name attribute is set to the name of the noun and the maximumSelectableOptions attribute is set to 1.

The following rule provides one example of transformation T2:

Rule 7: Each SelectionUIElement that only allows the selection of one Option is transformed into a Class associated with the Combobox Stereotype. The name of the class is composed of the name of the SelectionUIElement and "ComboBox".

Figure 4.6 illustrates the formalized Rule 7 for T2 in MOLA. It iterates over all SelectionUIElements where the attribute maximumSelectableOptions is set to 1 in the UI Specification and creates one class each. The name attribute of the class is set to the SelectionUIElement name to which the suffix "ComboBox" is added. It also creates an association with the corresponding stereotype.

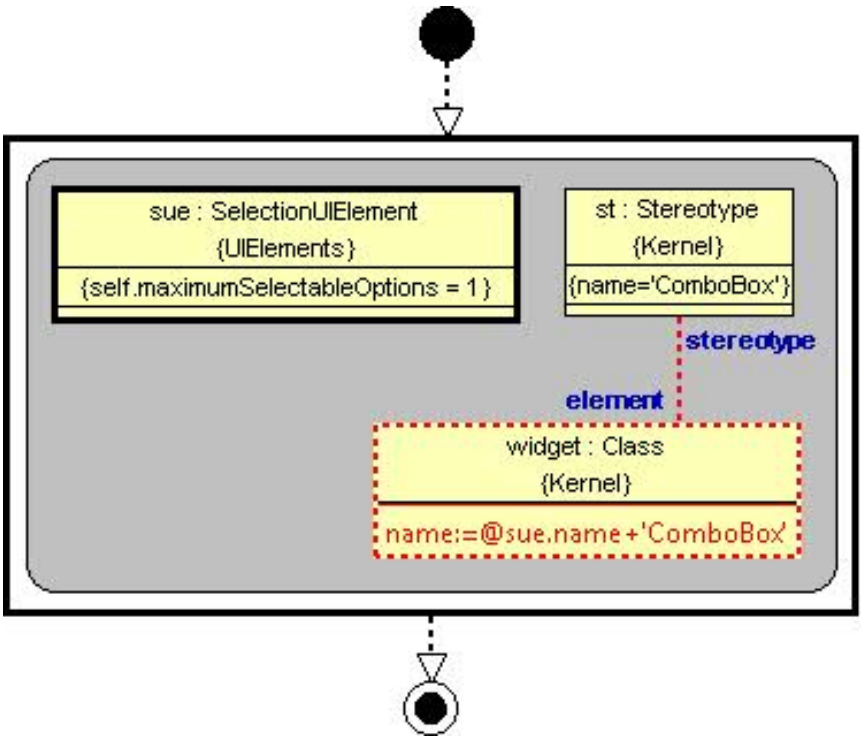


Figure 4.6: MOLA Rule R7 for T2.

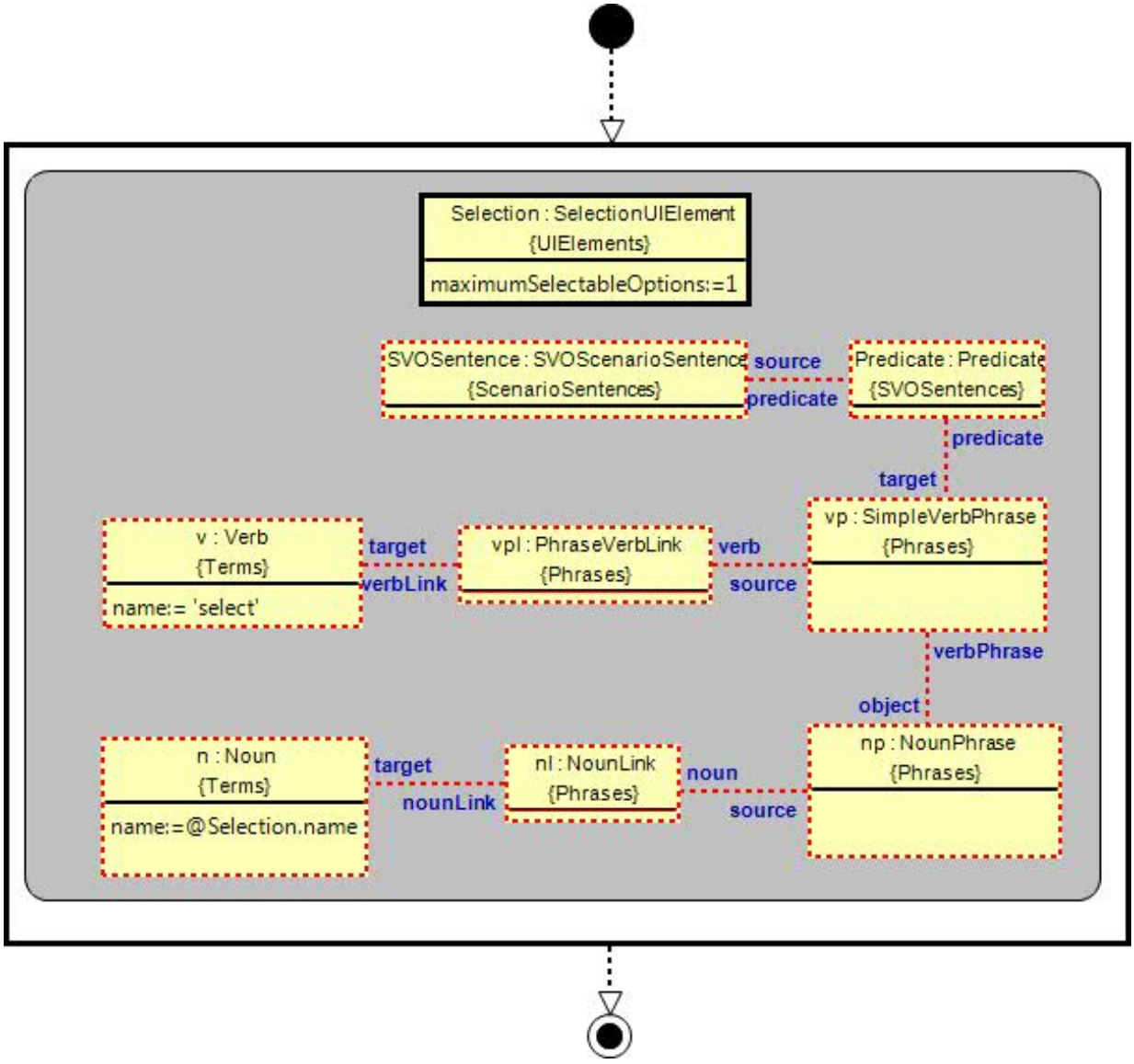


Figure 4.7: MOLA Rule R1' for T1'.

4.3 Transformations from UI Prototypes via UI Specifications to Requirements Specifications

This section introduces transformations for T2' and T1', from UI Prototypes via UI Specifications to Requirements Specifications.

The following rule for T2' represents the exact inverse to Rule 7 in T2:

Rule 7': Every class with the suffix "ComboBox" in the name attribute is transformed into a SelectionUIElement with the maximumSelectableOptions attribute set to 1.

The following rule for T1' represents the inverse to Rule 1 in T1 (but not exactly):

Rule 1': Every SelectionUIElement with the attribute maximumSelectableOptions set to 1 is transformed into an SVOScenarioSentence. The Verb "select" is the representative of the equivalence class for selection. The noun is the value of the name attribute of the SelectionUIElement. Figure 4.7 illustrates the formalized Rule 1' for T1' in MOLA.

There is a subtle difference with the inverse transformation, since Rule 1' always transforms to the verb "select". If another verb of the same type is used in the source model of T1, e.g., "choose", then applying T1 and subsequently T1' will not result in the identical model. This is more of theoretical than of practical interest, since transforming back and forth identically is not needed for applying this approach.

4.4 Deriving Inverse Transformation Rules Automatically

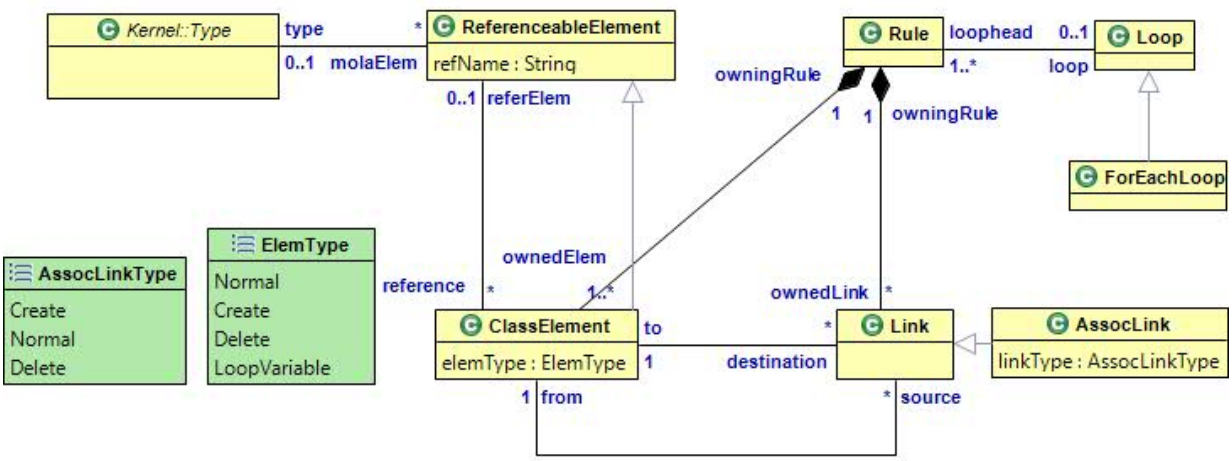


Figure 4.8: Excerpt of the MOLA Metamodel.

As explained in the previous section, the inverse transformation rules are needed to be able to transform automatically back to the Requirements Specification via UI Specification. Assume that a UI designer has specified a set of forward transformation rules in MOLA (e.g., the rules for T1 and T2). Instead of specifying the inverse transformation rules by hand, they may be

semi-automatically generated. This may reduce the manual effort and ensure consistency with the forward transformations.

For this purpose, I propose transformation *metarules* for transforming a rule to its inverse. These rules map metamodel elements of the transformation language to other metamodel elements of the same transformation language. So, these are mappings inside the same metamodel. Figure 4.8 shows an excerpt of the MOLA Metamodel which is used to define the Metarules described in this paper.

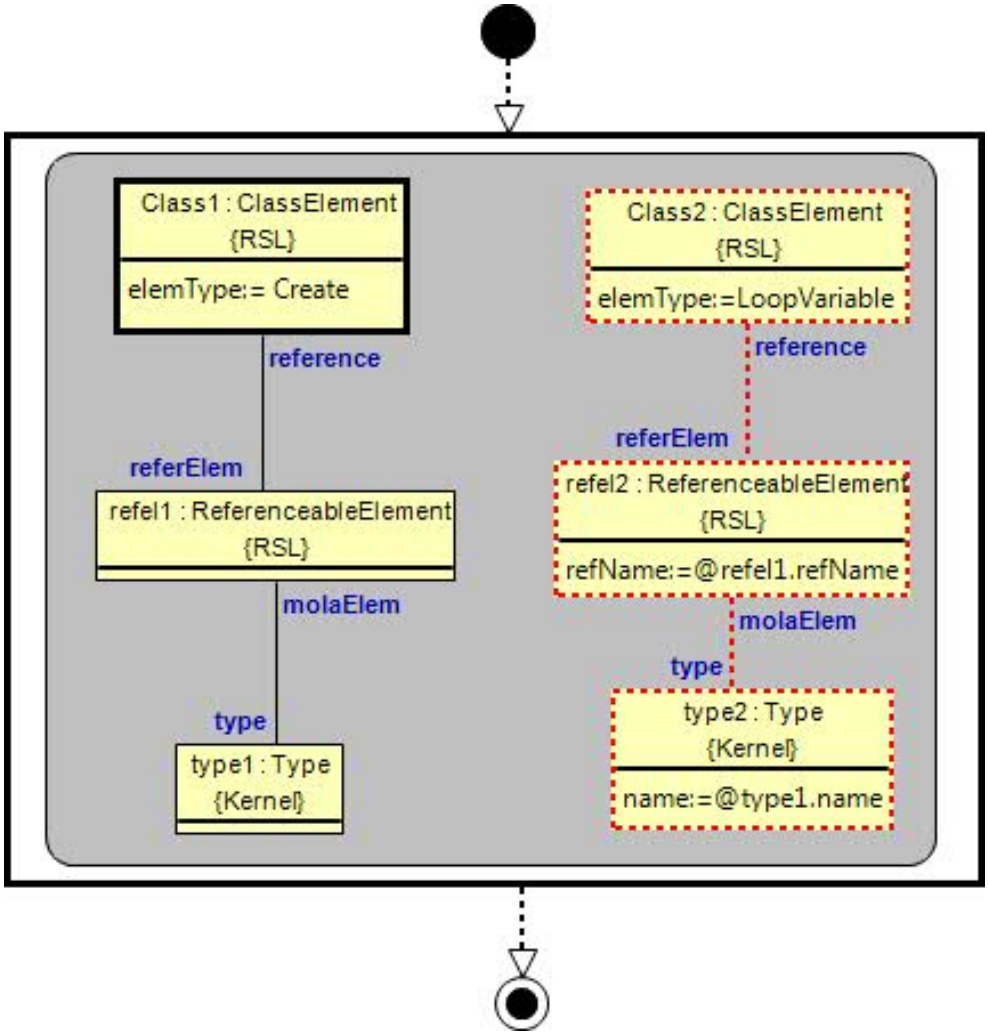


Figure 4.9: Metarule R1 in MOLA.

The following metarules transform T1 into T1' as well as T2 into T2':

Metarule 1: The "create" class is transformed into a "loop variable" class. For example, the created class "SelectionUIElement" in rule R1 in Figure 4.5 results in the "loop variable" class "SelectionUIElement" in Rule R1' shown on top in Figure 4.7. Metarule 1 is illustrated in MOLA in Figure 4.9. It iterates over all "ClassElement" classes which have the *elemType* set to "create". For each "ClassElement" class with *elemType* set to "loop variable", a "ReferenceableElement" class and a "Type" class is created. The *refName* attribute of the "ReferenceableElement" class refel2 is set to the value of the *refName* attribute of the refel1

class. The *name* attribute of the "Type" class type2 is set to the value of the *name* attribute of the type1 class. Two associations, one between Class2 and refel2, the other one between refel2 and type2 are created.

Metarule 2: The "loop variable" class is transformed into a "create" class. For example, the class "SVOScenarioSentence" in Rule R1 used to iterate over model elements is transformed into the create class "SVOScenarioSentence" in Rule R1', that is responsible for creating corresponding classes if rule R1' gets applied.

Metarule 3: The "normal" classes are transformed into "create" classes. For example, the other classes in Rule R1 in Figure 4.5 that define the pattern to match, get transformed into the create classes in Figure 4.7. Thus, the pattern will be also created during inverse transformations.

Metarule 4: The "normal" association link is transformed into a "create" association link. For example, the associations between the classes in Rule R1 that form the pattern to match, get transformed into the "create" association to relate the generated pattern classes.

Metarule 5: The "for each loop" is mapped on itself, and so is the "Rule". For example the outer bold rectangle (for each loop) and the rounded rectangle inside (Rule) in Rule R1 in Figure 4.5 are transformed to the outer bold rectangle and the rounded rectangle inside in Rule R1' in Figure 4.7.

Applying these metarules to the MOLA Rule R1 for T1 results in the rule R1' for T1'. By applying Metarule 1, the element type of the SelectionUIElement class is changed to "loop variable". By applying Metarule 2, the element type of the SVOScenarioSentence is changed to "create". By applying Metarule 3, the element type of the classes linked to the SVOScenarioSentence class is changed to "create". By applying Metarule 4, the association link type of the associations between the classes connected to the SVOScenarioSentence is changed to "create". By applying Metarule 5, the outer bold rectangle (for each loop) and the rounded box (rule) is mapped to itself. The same metarules can transform the MOLA Rule R7 for T2 to the rule R7' for T2'. The execution order of these metarules has no impact on the result.

This approach for deriving inverse transformation rules has some limitations and poses some restrictions on the design of the forward transformation rules. The mapping between the source and target model have to be bijective. Inverse transformation rules can only be derived automatically, if only one class is created in the MOLA rule. Otherwise the loop variable needed by MOLA could not be uniquely identified anymore. Further, multiple rules referring to same model elements in their pattern lead to inverse transformation rules creating these model elements more than once, and thus, require a merging of the created model elements. Some information in the inverse transformation rule cannot be easily derived automatically, but can be added by heuristics. For example, the name of the verb in Rule R1' in Figure 4.7 cannot be automatically derived by general metarules from the "UISelectionElement". However, it can be set by post-processing the generated inverse transformation rules with domain-specific heuristics.

The advantages of deriving inverse transformation rules automatically are:

- Rule changes in the forward transformation can be easily kept consistent with the backward transformation.
- This approach reduces the manual effort of creating inverse transformation rules by hand.

5 Comparison between a Declarative and a Procedural Transformation Language

This chapter compares a *declarative* transformation language, the discourse transformation language (DTL) with a *procedural* transformation language, the MOdel transformation LAnguage (MOLA) based on the model transformation approach features defined in [CH06]. DTL belongs to the category of structure-driven model transformation approaches whereas MOLA belongs to the category of graph-transformation-based approaches. DTL uses declarative rules whereas MOLA uses procedural rules. The same transformation rules are expressed in DTL and MOLA for comparison. Therefore some transformations in DTL, which have been used in Chapter 3 to transform Discourse Model elements to Structural UI Model elements have also been implemented in MOLA. Furthermore, the differences between MOLA and DTL are explained. This chapter also contains the experiences made during the application of DTL and MOLA.

5.1 MOLA vs. DTL

This section gives a comparison between MOLA and DTL regarding the main model transformation language features. These are location determination, rule scheduling, rule organization, source-target relationship, incrementality, directionality, and tracing. Additionally, the transformation rules specified in both languages are compared.

Location determination:

Location determination is the strategy for determining the model locations to which transformation rules are applied. We distinguish between three different kinds of location determination strategies: deterministic, non-deterministic and interactive. The location determination strategy used in MOLA and DTL is deterministic due to the fact that transforming one source model twice will result in the same target model. DTL uses the standard traversal strategy depth first to traverse the containment hierarchy in the source model. In MOLA, a local search plan generation based algorithm [Sos10] is used to find the instances in the source model.

Rule Scheduling:

Form: In MOLA the form of rule scheduling can be classified as explicit internal scheduling because a transformation rule can directly invoke other rules. In contrast, DTL uses implicit scheduling. That implies that the user has no explicit control over the scheduling algorithm. The

only way to influence the rule execution order is by specific rule design. The rule scheduling is hidden from the user in the transformation engine.

Rule selection: In MOLA transformation rules are selected by an explicit condition. In contrast DTL offers a conflict resolution mechanism which is based on four criteria. First the transformation rules are selected appropriate to the device specification. Second they are ranked according to their specialization degree, from the most specific to the least specific. In the case of more than one rule being the most specific, these are ranked again according to the estimated space the target pattern of the rule would occupy on the screen of a GUI. Finally, if there is still no unique decision possible the rule with the highest priority gets applied. If rules have equal priority the decision about which rule gets applied is not deterministic.

Rule iteration: In the literature three different kinds of rule iteration mechanisms are known. Recursion, looping and fixpoint iteration. In MOLA rule iteration is supported by the use of a looping construct, whereas in DTL rule iteration is not supported.

Phasing: In MOLA rule scheduling with separated phases is not supported, whereas DTL allows to separate the transformation process in two phases, with each phase having a specific purpose, and only certain rules can be invoked in a given phase. The first phase creates the containment hierarchy with placeholders in the target model and the second phase replaces them according to the content attributes and references. This is done to achieve a separation of concerns. The first phase is responsible for the rendering of Discourse Model elements. The second phase takes care of the rendering of the Communicative Acts content.

Rule Organization:

Both MOLA and DTL allow the packaging of rules into modules. MOLA offers a way to define a rule based on one or more other rules. Thus it allows the reuse of rules by their logical composition to build new rules. MOLA does not support rule inheritance. DTL does not support any reuse mechanism. The organizational structure of rules in MOLA is independent of the target or source language, whereas in DTL it is source oriented.

Source-Target Relationship:

In DTL it is mandatory that a new target model is created for each transformation execution cycle. In contrast, MOLA supports the update of an existing target model. It allows an In-Place Update as well as destructive updates. “In-Place Update” means that the original source model can be modified by the transformation rules. “Destructive update” means that a transformation rule can delete a model element in the target model.

Incrementality:

Incrementality is the ability to update an existing model based on changes in another model. In DTL there is no support for Incrementality, whereas in MOLA Target Incrementality is partly supported by the use of traceability links. Updating the target model after modifying existing elements in the source model is not supported. However, updating the target model after new elements have been added to the source model is supported. The preservation of user edits in the target model is partially supported by the use of traceability links. Source Incrementality is not supported in MOLA.

Directionality:

DTL only supports the creation of unidirectional transformations. This means that the transformation rules can only be executed in one direction, e.g. from the Discourse Model to the

Structural UI Model. MOLA, however, supports the creation of bidirectional transformations by defining two separate complementary unidirectional rules, one for each direction.

Tracing:

Both in DTL and MOLA, the creation of traces is done by inserting traceability links manually. In MOLA tracing information has to be created as a target element. In DTL tracing can be defined in the property of the target elements. In DTL the storage location of the traceability links is only the target model whereas in MOLA they can be both source and target model.

5.1.1 General Transformation Rule Comparison

Some of the transformation rules specified in DTL, which have been used in Chapter 3 to transform the Discourse Model to the Structural UI Model, are also implemented in MOLA for comparison. However, only the transformation rules that directly correspond to one of the DTL rules have been implemented in MOLA. The main MOLA rule which is needed to execute the other defined MOLA rules in a specific order is skipped because there is no corresponding DTL rule to compare with. Therefore the main difference between the DTL and the MOLA rules is that many extra transformation rules in MOLA have to be defined to cover the same features that the DTL engine can offer out of the box, like conflict resolution. The name of the transformation rules compared relate to the matching source element of the rule e.g., the rule that matches the Background RST relation of the Discourse Model is called Background Rule.

Figure 5.1 illustrates the Background Rule (specified in Section 3.5) formalized in MOLA, composed of the following elements. The outer bold rectangle symbolizes a *for-each* loop. The rounded rectangle inside represents the actual rule that will be repeated for each matched element. The small boxes inside the rule represent different kinds of classes, depending on their borderline style. When the thickness of the border line is regular, they represent a "normal" class. A bold border lined box represents a loop variable. A dashed lined box border represents a class that will be created by the transformation rule. The small black circle represents the starting point of the rule. The double rounded circle represents the end point of the rule. In particular, the Background rule iterates over all Background RST relations. Whenever a Background relation connects a nucleus tree and a satellite tree, the rule matches. In this case, a Root panel containing a panel for the satellite branch and the nucleus branch is generated in the structural UI model. The Root panel is linked to a container panel, which is the input parameter of the rule. For the root panel a GridLayout and a Style element is created. For each nucleus and satellite panel a GridLayoutData element are created. Next the execution is handed over to the rule responsible for the nucleus and subsequently to the rule transforming the satellite. Both rules get the panel as a parameter, so that they can add their results to the panel.

Figure 5.2 illustrates the Background Rule specified in DTL. The source pattern is defined in the middle of the rule as a Background relation with a nucleus and satellite branch. The target pattern is defined below and defines a panel composed of a Style element, GridLayout element and two panels as placeholders for the nucleus and satellite branch. Each composed panel has a GridLayoutData element. The Mapping between the nucleus and satellite link of the source pattern and the subpanels of the target pattern are defined at the beginning of the rule.

Figure 5.3 illustrates the Adjacency Pair Rule specified in Section 3.5 formalized in MOLA. It iterates over all Adjacency Pairs of the Discourse Model and creates a Panel each. It also creates an association to the parent panel which the rule gets as a parameter.

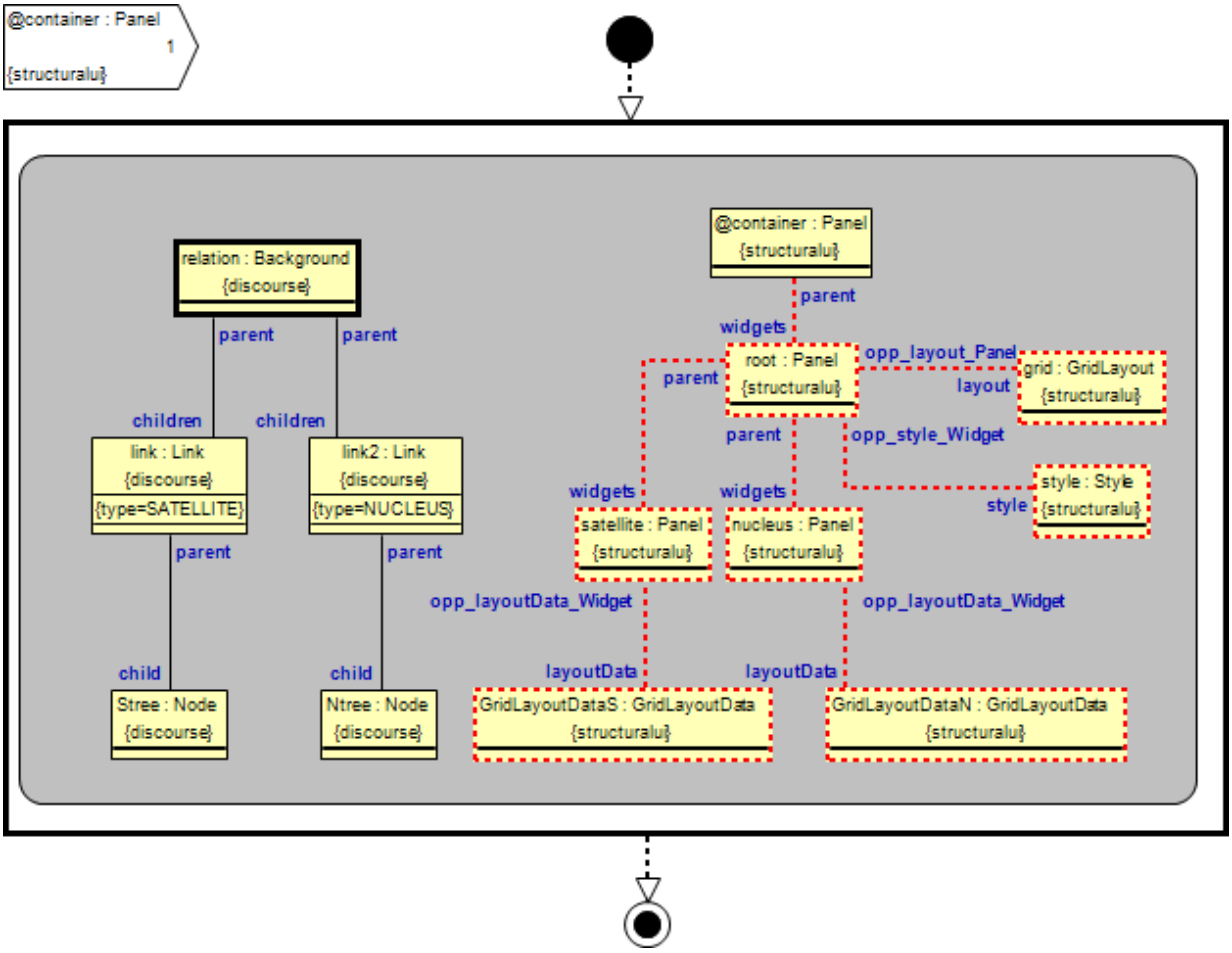


Figure 5.1: Background Rule specified in MOLA

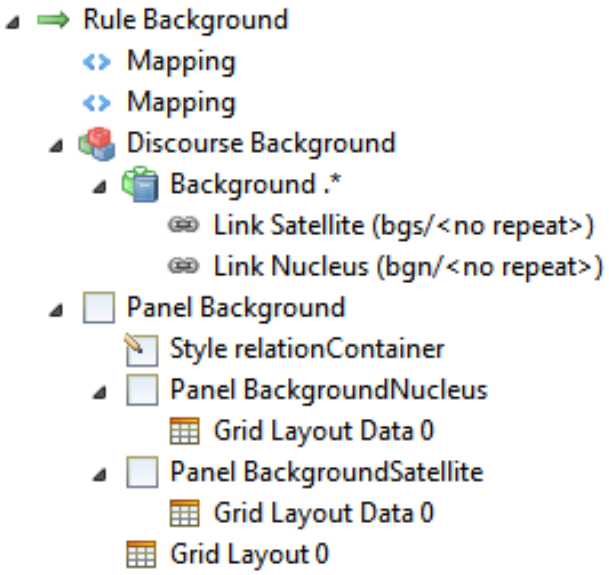


Figure 5.2: Background Rule specified in DTL

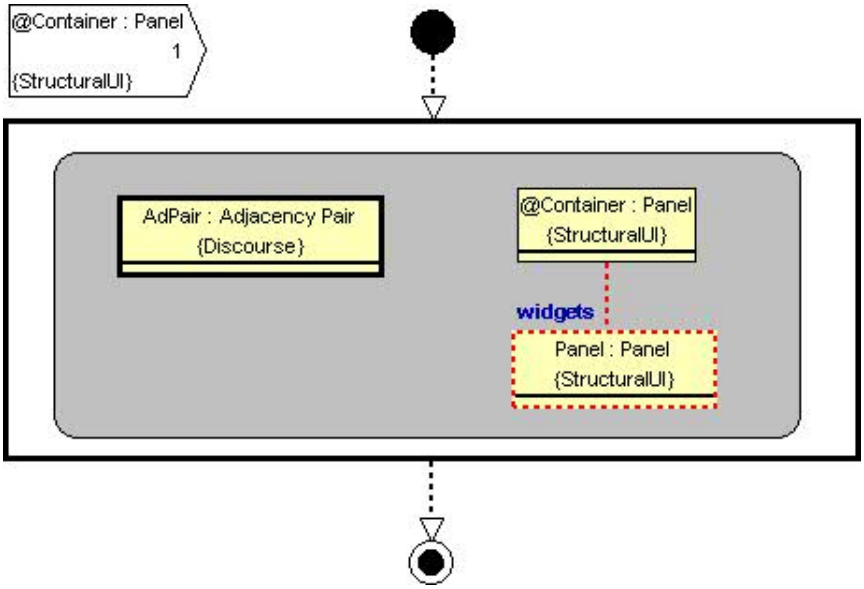


Figure 5.3: Adjacency Pair Rule specified in MOLA

Figure 5.4 illustrates the Adjacency Pair Rule specified in DTL. The source pattern defines an Adjacency Pair. The target pattern specifies a panel that is created for each Adjacency Pair.

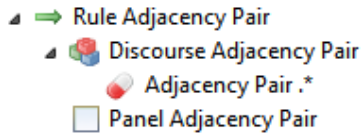


Figure 5.4: Adjacency Pair Rule specified in DTL

Figure 5.5 illustrates the Offer-Accept Rule specified in Section 3.5 formalized in MOLA. It iterates over all *Offer*s that are associated with an *Accept* via an Adjacency Pair. The parameter container of type panel represents the parent panel and allows the rule to add the generated widgets to the panel. Depending on the type of the communicative act's content, one of two alternative rules is selected. If the content is a kind of List (consisting of more than one element) a *ListWidget* and a *Button* are generated. In all other cases only a *Button* is generated in the Structural UI Model.

Figure 5.6 illustrates the Offer-Accept Rule specified in DTL. The source pattern matches only Offer-Accept Adjacency Pairs which have a multiple cardinality defined for the content of the Offer Communicative Act. The target pattern creates a ListWidget with a button.

Domain:

A Domain is the part of a rule responsible for accessing either the source or the target model. It has an associated domain language specification that describes the possible structures of the models for that domain. The domain language both in MOLA and DTL is the Meta Object Facility (MOF). In a DTL rule, the in-domain (source) is specified with the elements contained in the Discourse element illustrated in the upper part of the rule in Figure 5.2. The out-domain (target) is specified with the elements contained in the lower compartment of the rule illustrated in the lower part of Figure 5.2. In this rule the Lower compartment starts with the element "Panel Background". In a MOLA rule, as illustrated in Figure 5.1, all elements with a regular and

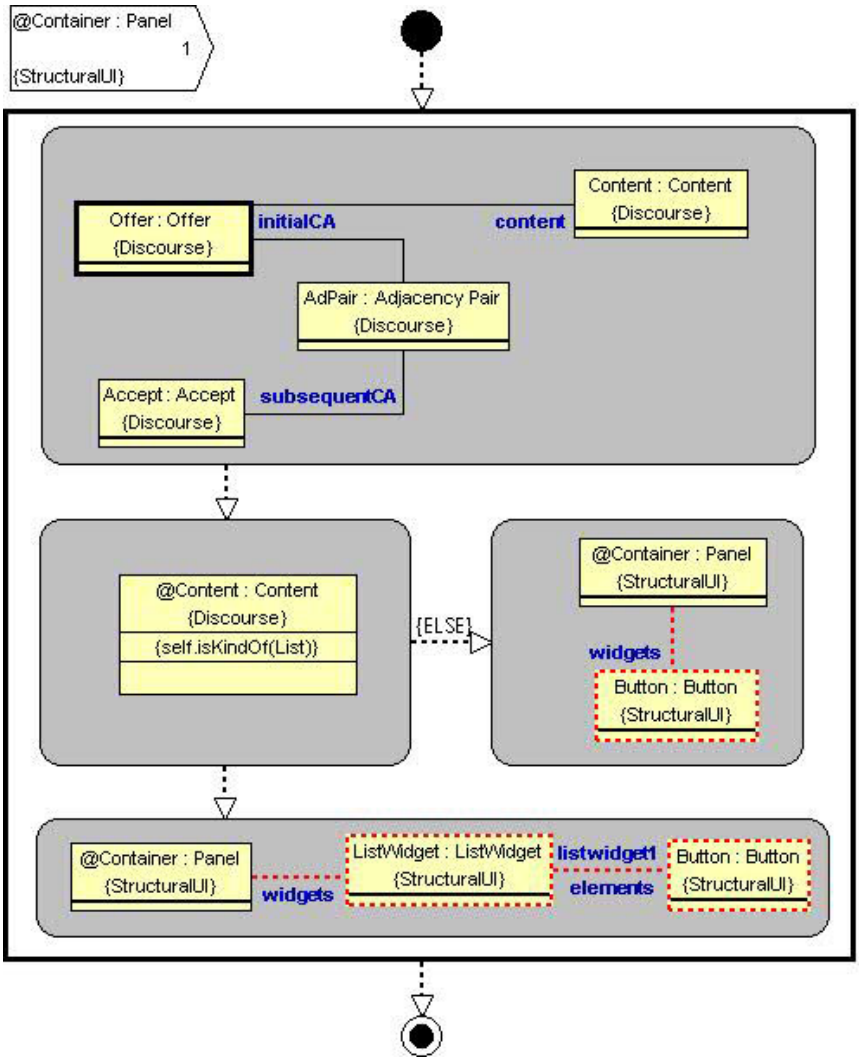


Figure 5.5: Offer-Accept Rule specified in MOLA

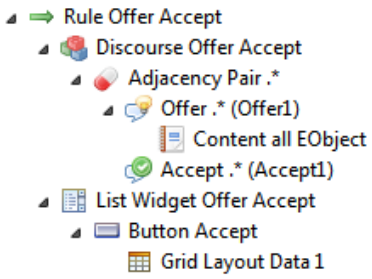


Figure 5.6: Offer-Accept Rule specified in DTL

bold line strength belong to the in-domain, whereas all elements with a dashed line belong to the out-domain. Thus, in a MOLA rule there is no local separation between the elements of the in-domain and the out-domain. In contrast in a DTL rule, the in-domain and out-domain elements are placed in separate compartments. MOLA and DTL rules use graphs to present the structure of their source and target patterns. Both represent their patterns with a concrete graphical syntax illustrated in Figure 5.1 and Figure 5.2 respectively. MOLA rules have a procedural language paradigm whereas DTL rules are based on logic programming. Values are specified in MOLA and DTL rules through constraints. MOLA rules create elements explicitly but DTL rules implicitly by the framework. MOLA rules are typed semantically as well as DTL rules.

Syntactic Separation:

DTL rules have a left hand side (LHS) operating on the source model separated from the right hand side (RHS) operating on the target model. In Figure 5.2 the separation between the LHS, contained in the Discourse element and the RHS, starting with the "Panel Background" is visible. In contrast, MOLA rules have no syntactic separation. In Figure 5.6 it can be seen that elements that are created (dashed line) belonging to the target model are not separated from elements from the source model.

Multidirectionality:

MOLA and DTL rules do not allow the execution in both directions. Their in-domain is clearly separated from the out-domain whereas rules supporting multidirectionality are defined over in/out-domains.

Application condition:

Transformation rules expressed in DTL can have constraints that serve as an application condition. This condition must return "true" in order for the rule to be executed. Transformation rules in MOLA do not support an application condition.

Intermediate structure:

The execution of MOLA and DTL rules does not require the creation of intermediate structures which are not part of the models being transformed.

Parameterization:

Control parameters, generics and higher-order rules are neither supported by MOLA nor by DTL rules.

Reflection and Aspects:

Reflective access to transformation rules during the execution of transformations is not supported in MOLA and DTL. The definition of *aspects* to express concerns that crosscut several rules are not supported in MOLA and DTL.

5.1.2 Concrete Rule Example Comparison

Comparing the Background rule specified in MOLA in Figure 5.1 with the Background rule specified in DTL in Figure 5.2, there are two difference. In the Background rule specified in MOLA an input parameter that serves as a container for the created elements has to be specified whereas in DTL mappings have to be defined to map the link satellite and the link nucleus to dedicated panels.

Comparing the Adjacency Pair rule specified in MOLA in Figure 5.3 with the Adjacency Pair rule specified in DTL in Figure 5.4 there is only one difference. The Adjacency Pair rule in MOLA additionally needs a container panel as a parameter where the created panel can be inserted.

Comparing the Offer-Accept rule specified in MOLA in Figure 5.5 with the Offer-Accept rule specified in DTL in Figure 5.6 there are some differences. Like the Adjacency Pair rule in MOLA, the Offer-Accept rule in MOLA additionally needs a container panel as a parameter where the created panel can be inserted. In the Offer-Accept rule in MOLA a condition is modeled to decide whether only a button should be added to a panel or a ListWidget containing a button. The Offer-Accept rule specified in DTL in Figure 5.6 only covers the ListWidget containing a button case. An equivalent second rule is needed in which the cardinality of the content of the Offer is set to "one". In contrast to the Offer-Accept rule specified in MOLA in Figure 5.5, the decision does not have to be modeled explicitly. The framework of the transformation language takes care of it.

The most important difference between the three rules in DTL and in MOLA is that for the execution of the MOLA rules an additional MOLA rule is needed that defines the execution order. In DTL the execution order is handled by the framework automatically.

5.1.3 Personal Experience with DTL and MOLA

The personal experience made is that writing rules in MOLA is connected with more effort for the designer than in DTL. This is due to the fact that the designer has to keep the execution order in mind when designing the rules whereas in DTL the designer can fully concentrate on the transformation rule design itself. Conditions have to be explicitly modeled in MOLA whereas in DTL they are handled by the framework. Many features, e.g. conflict resolution have to be implemented in MOLA in form of additional rules, which makes the design task much more complicated. DTL however, hides the complexity much better from the user by supporting many features out of the box and might, therefore, be even suitable for the end user. However, MOLA gives more influence on the transformation process and can, therefore, be more suitable for expert users.

5.2 Differences between MOLA and DTL

- **MOLA provides graphical representation of transformations:**

  MOLA provides an easily readable graphical transformation language by combining traditional structured programming in a graphical form (a sort of "structured flowcharts") with pattern-based rules. This is achieved by introducing a graphical loop concept, augmented by an explicit loop variable. The loop elements can easily be combined with rule patterns. Other structured control elements are introduced in a similar way.

- **MOLA allows explicit control of rule scheduling:**

  A consequence of the procedural nature of MOLA is the explicit control of rule scheduling. The user can explicitly decide which rule should be executed and when, by defining the concatenation order. Thus expert users have full control of the rule scheduling and can design their rules specifically for their desired rule execution sequence.

- **DTL allows multiple rules to match the same source pattern:**

  In DTL one source pattern can be matched by multiple rules. Therefore it has to provide a conflict resolution mechanism based on the device specification, the rule specialization, screen space and priority. In model-driven GUI development this feature is of major importance because one abstract concept can be realized on the GUI in different ways. Thus DTL is specially tailored for the GUI development based on the Discourse and Structural UI Model and provides all necessary features.

- **MOLA is suitable for general purpose transformations:**

  In contrast to DTL, MOLA is not designed for any specific transformation purpose. Therefore it can be used to transform any source model to any target model. If specific features are needed, they have to be implemented in the form of MOLA rules.

- **DTL handles the rule execution order automatically:**

  In DTL the rule execution order is handled automatically by the framework. Therefore the designer can fully concentrate on writing the transformation rules. Rules can be designed independently of other rules. This makes rule design much more convenient for the designer by reducing the cognitive load. This is a key benefit of any declarative transformation language and it reduces the effort for the rule creation tremendously.

5.3 Comparison Summary

This section gives a summary of the comparison between the transformation languages MOLA and DTL. Therefore, two tables are presented which show the supported features. Table 5.1 illustrates the comparison between the supported transformation language features of MOLA and DTL. The boxes filled with a cross symbolize that this feature is supported by the particular transformation language. Table 5.2 illustrates the transformation rule features supported from MOLA and DTL.

DTL is a declarative transformation language specifically developed for transforming Discourse Models to Structural UI Models. The characteristic feature is multiple rule matching, which is of major importance for GUI development. Thus it allows one source pattern to be associated with different target patterns, more specifically Structural UI Model parts. It also provides a sophisticated conflict resolution mechanism tailored specifically to GUI development.

MOLA is a procedural transformation language, which is not restricted to a specific source or target metamodel. The characteristic feature is the graphical representation of the transformation rules which help the designer to define rules. It allows the explicit control of the rule scheduling due to the procedural nature of the transformation language.

In contrast to DTL, which is specifically suitable for GUI generation, MOLA is suitable for any model-to-model transformation purpose. However MOLA is better suited for expert users because it allows them to have more influence on the transformation process, whereas DTL is more end user friendly because it allows the designer to focus on the development of the transformation rules.

	MOLA	DTL
LOCATION/DETERMINATION		
deterministic	X	X
RULE SCHEDULING		
Form:		
implicit		X
explicit internal	X	
Rule Selection:		
explicit condition	X	
conflict resolution		X
Rule Iteration		
looping	X	
Phasing		X
RULE ORGANISATION		
modalarity mechanism	X	X
Reuse Mechanism:		
logical composition	X	
Organisational Structures		
source oriented		X
independent	X	
SOURCE TARGET RELATIONSHIP		
new target		X
existing target > inplace or destructive update	X	
INCREMENTALITY		
target incrementality	X	
preservation of user edit in target	X	
DIRECTIONALITY		
unidirectional		X
multidirectional	X	
TRACING		
manual creation	X	X
Storage Location:		
target model	X	X
source model	X	

Table 5.1: Transformation language feature comparison between MOLA and DTL

	MOLA	DTL
DOMAIN		
Domain Language		
Structural UI-Model		X
Discourse Model		X
Any Model	X	
Static Mode		
in/out	X	
in		X
out		X
Body		
patterns	X	X
- stucture graphs	X	X
- concrete graphical syntax	X	X
logic		
- language paradime		
- logic		X
- procedual	X	
- value specification		
- constraint	X	X
- element creation		
- explicit	X	X
Typing		
semantically typed	X	X
SYNTACTIC SEPERATION	X	X
MULTIDIRECTIONALITY	X	
APPLICATION CONDITION		X

Table 5.2: Transformation rule feature comparison between MOLA and DTL

6 Feasibility Study CommRob: GUI Generation for a Given Finger-based Touchscreen GUI Design

This chapter presents a feasibility study, which shows how the discourse-based WIMP UI generation approach was used to generate the GUI for a given finger-based touchscreen GUI design. In particular it shows the tailoring of the transformations between Discourse Models and Structural UI Models for a given finger-based touchscreen GUI design. Therefore, the correspondent's between the Discourse Model and the resulting GUI is shown. The main challenges that had to be dealt with are: Adapting the layout according to a given design, presenting content according to a given design and customizing style according to a given design. In the CommRob project, the discourse-based WIMP UI generation approach, presented in Chapter 3, was used to generate the GUI for the finger-based touchscreen of a Shopping Trolley. The Discourse Models specified the communication between the Trolley and the user. This chapter contains some discourse models of the CommRob project and the resulting GUI [BEF+10, EFK+10a], the CommRob GUI generation challenges as well as lessons learned.

6.1 The CommRob Discourse Models and the Corresponding GUI

The GUI for the CommRob Shopping Trolley has been developed based on a given finger-based touchscreen GUI design. We tried to capture the interaction using our Discourse Models. Subsequently, we used these Discourse Models to automatically generate large parts of the GUI. To improve the usability of the GUI, we added additional information to the transformation process (e.g., style and layout) and we included manually written components.

Figure 6.1 shows the start screen of the CommRob GUI. It is composed of five areas. The Shopping Discourse (1), the Status Area (2), the Map Widget (3), the Navigation Widget (4) and the Help Area.

The GUI that has been generated based on the approach presented in Chapter 3 is displayed in Area 1 of Figure 6.1. This area is the only finger-based touchscreen GUI part which is generated with the transformataion rules presented in Chapter 3 and contains all elements that allow the user to interact with the application. All other components on the screen are not generated from Discourse Models and are only used to display information for the user. The most universal instrument to do so offers Area 2, the so-called Status Area. It is used to inform the user about different

types of events. Such events reflect the status of the application (e.g., "Managing ShoppingList"), they contain information from the Trolley layer (e.g., "bumper hit") or they convey information from another Trolley (e.g., "Your partner is going to checkout"). The MapWidget displayed in Area 3 shows a map of the supermarket and informs the user about the Trolley's actual position. The NavigationWidget in Area 4 is used to inform the user in which direction the robot is about to move. Area 5 displays information that should help the user during his shopping experience. This area is updated according to what is displayed in Area 1. All other widgets that can be seen in Figure 6.1 (i.e., the clock, the mute button and the exit button) are not related to the application logic and, therefore, not to any discourse. Thus they are also not relevant for the transformations like area 2,3,4 and 5.

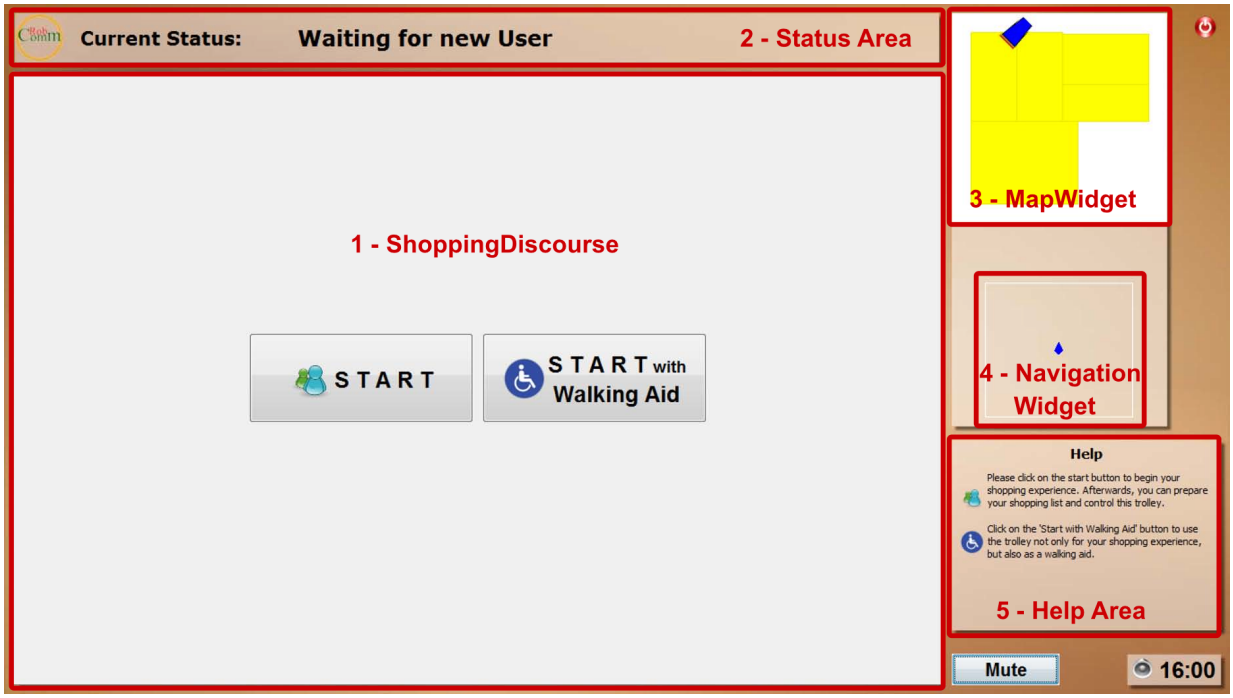


Figure 6.1: CommRob Start Screen

Figure 6.2 illustrates the root of the shopping Discourse Model. This model specifies the communication between two communication parties, the Trolley and the Customer. They are represented in the diagram with actor symbols in the upper left. The shopping Discourse Model is used to generate the GUI presented in the Shopping Discourse (1) Area in Figure 6.1. The discourse starts with the alternative of two requests either to Start shopping or to start shopping with the help of the walking aid [EKA+11]. The left button is generated to send the Request to Start Communicative Act and the right button to send the Request to Start with Walking Aid. The Alternative relation results in the parallel presentation of the possibility to send each of the requests. After the Trolley has been successfully attached, the second nucleus branch of the sequence becomes active. The Tree branch of the lower IfUntil is executed if the Trolley is not moving whereas the Then branch is executed if the Trolley is moving. The Else branch becomes active when the payment is in progress. Finally, when the user completed the payment, the Then branch of the upper IfUntil becomes active and the Informing that the Trolley has returned gets uttered.

Figure 6.3 illustrates the subpart of the shopping Discourse Model which is linked to the Tree branch of the lower IfUntil in Figure 6.2. It specifies all the interaction when the Trolley is not

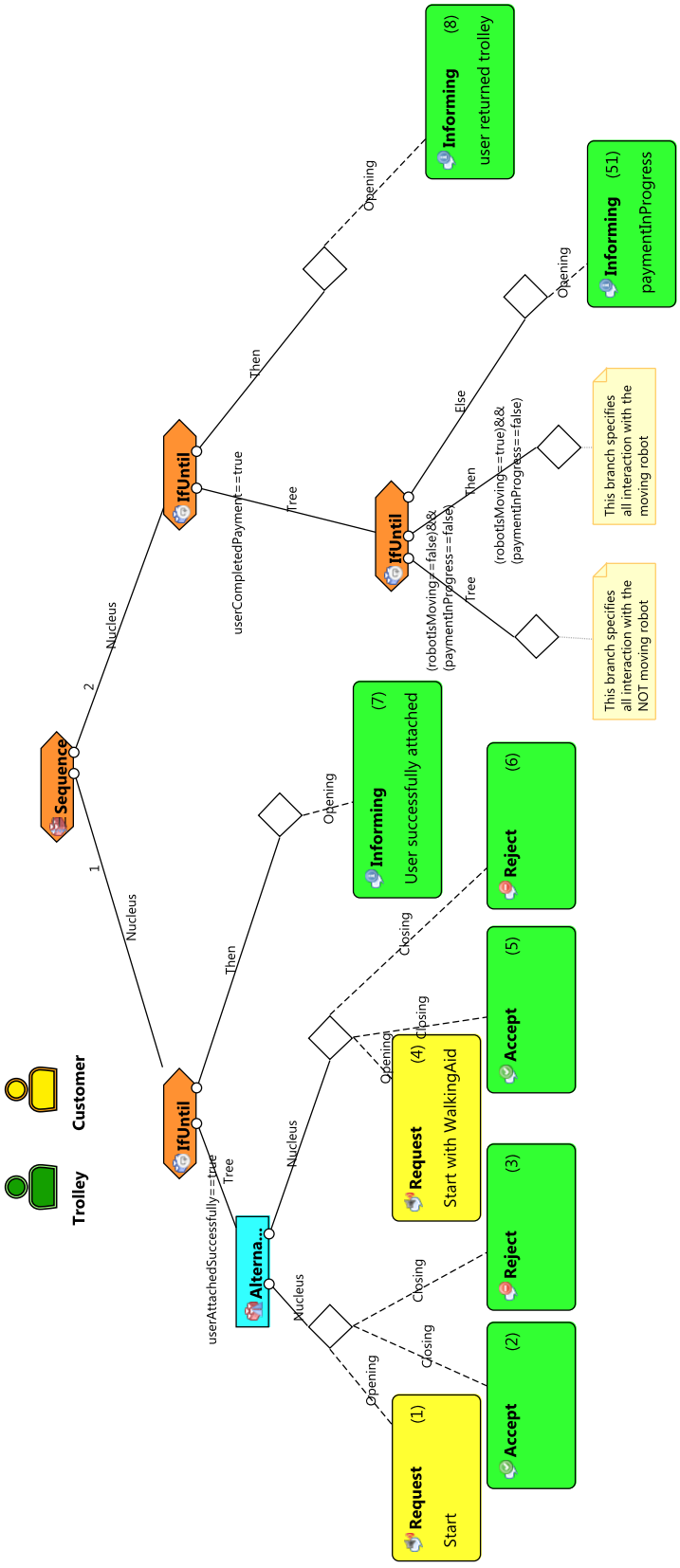


Figure 6.2: Shopping Discourse Root

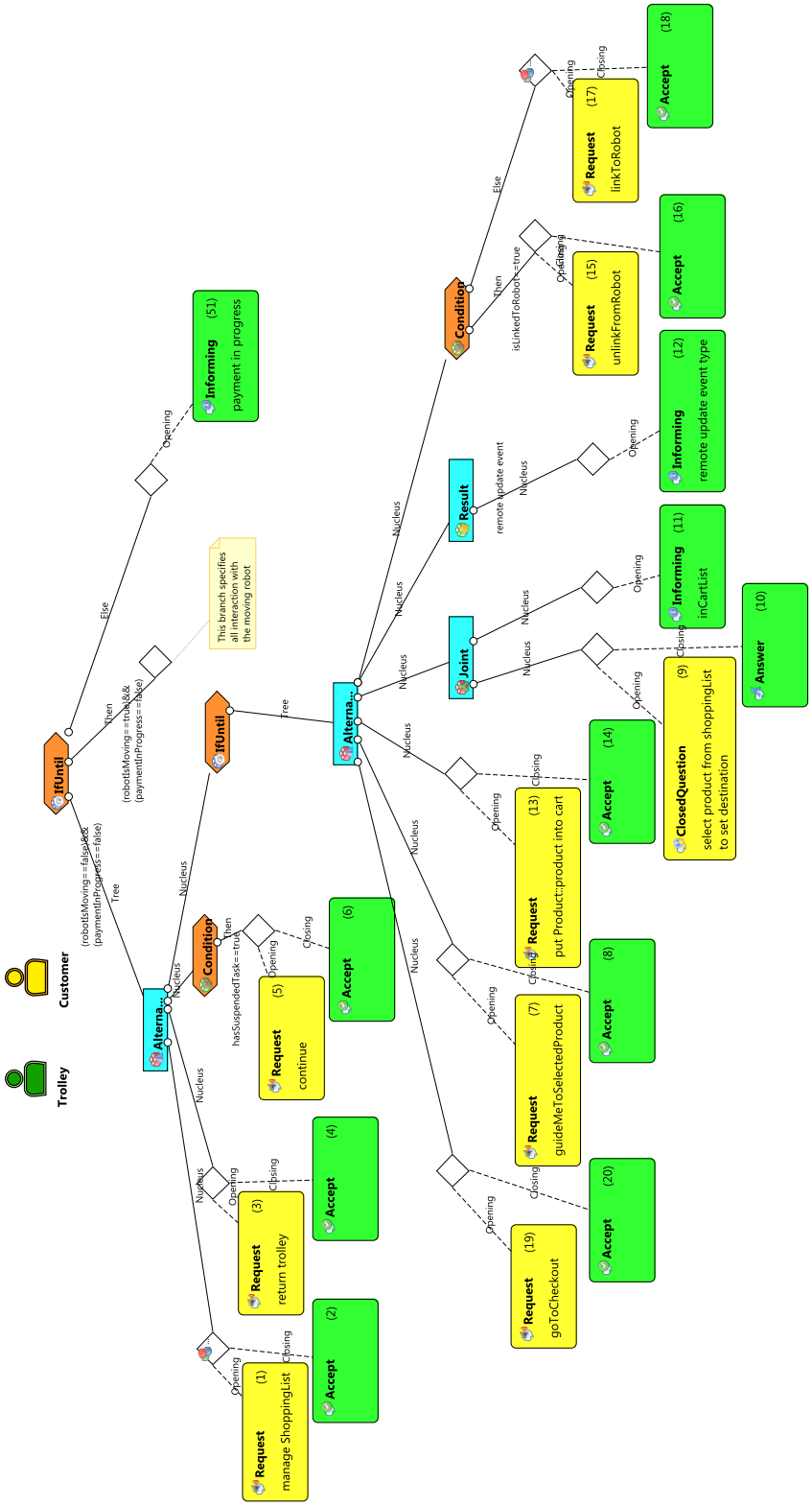


Figure 6.3: Shopping Discourse – Robot is NOT MOVING branch

moving. The user has the alternative to send a Request for managing the shopping list, to return the Trolley, to continue the suspended task, to go to the checkout counter, to guide her to a selected product, to put a product into the Trolley, to select a product from the shopping list as the next destination and to link or unlink with another Trolley. The resulting GUI screen is illustrated in Figure 6.4. The area marked with the number 4 results from the lower Alternative subtree. The area marked with the number 1 results from the request to manage shopping list on the left. The area marked with number 2 results from the request to continue in the middle. The area marked with number 3 results from the request to return Trolley. The Condition relation linked to the upper alternative is used to restrict the utterance of the Request – Accept to a condition.

An IfUntil whose Tree branch has no condition is, in fact, an endless loop, which can only be ended by one of its parent relations. The IfUntil relation which is linked to both alternatives in the middle of the diagram is needed to loop over the alternative linked to the Tree branch. The loop stops if the condition of the Tree branch of the root IfUntil is not fulfilled anymore. More precisely, if the robot leaves the NOT MOVING state (i.e., the robot starts moving or the payment is in progress), because this implies that the repeat condition of the Tree branch is not fulfilled anymore.

The nucleus branch of a result relation is executed as soon as its condition is fulfilled. The result relation in Figure 6.3 is used to inform the user about events coming from the linked Trolley.

The Request – Accept adjacency pair for putting a product into the Trolley is not rendered on the GUI. This adjacency pair is rendered by the bar code reader and sent when a product is scanned.

Figure 6.5 illustrates the subpart of the shopping discourse which models the interaction when the Trolley is moving. It is linked to the Then branch of the lower IfUntil relation in Figure 6.2. The Trolley has the alternative either to inform about the shopping list, destination list and the In Cart List or to inform about the updated task and status. The user has the alternative to request the Trolley to stop or to inform that she has put a product into the Trolley. This Informing is also not rendered on the GUI but by the barcode reader. This signifies that scanning a product is possible while the robot is moving as well as while it is not moving.

Figure 6.6 illustrates the inserted sequence for managing the shopping list. It is embedded in the adjacency pair Request – Accept to manage the shopping list represented in the left of Figure 6.3. The user has the alternative either to select one product category and then to elaborate on it by selecting a product of the selected category, or to select a product to remove from the shopping list, or to request the finishing of the shopping list management. The IfUntil is used to loop over the alternative, until the customer requests to finish the inserted sequence. The Result relation is used to model the events coming from the linked Trolley.

Figure 6.7 illustrates the inserted sequence for linking to another Trolley. It is embedded in the adjacency pair Request – Accept to link to another Trolley represented in the right of Figure 6.3. The user has the alternative to either select one robot to link or to request the cancellation for linking to another Trolley.

6.2 The CommRob GUI Generation Challenge

One of the challenges in the CommRob project was to fully-automatically generate part of the GUI for a finger-based touchscreen according to a given GUI design. This means that the discourse-

Figure 6.4: CommRob screen with customized layout

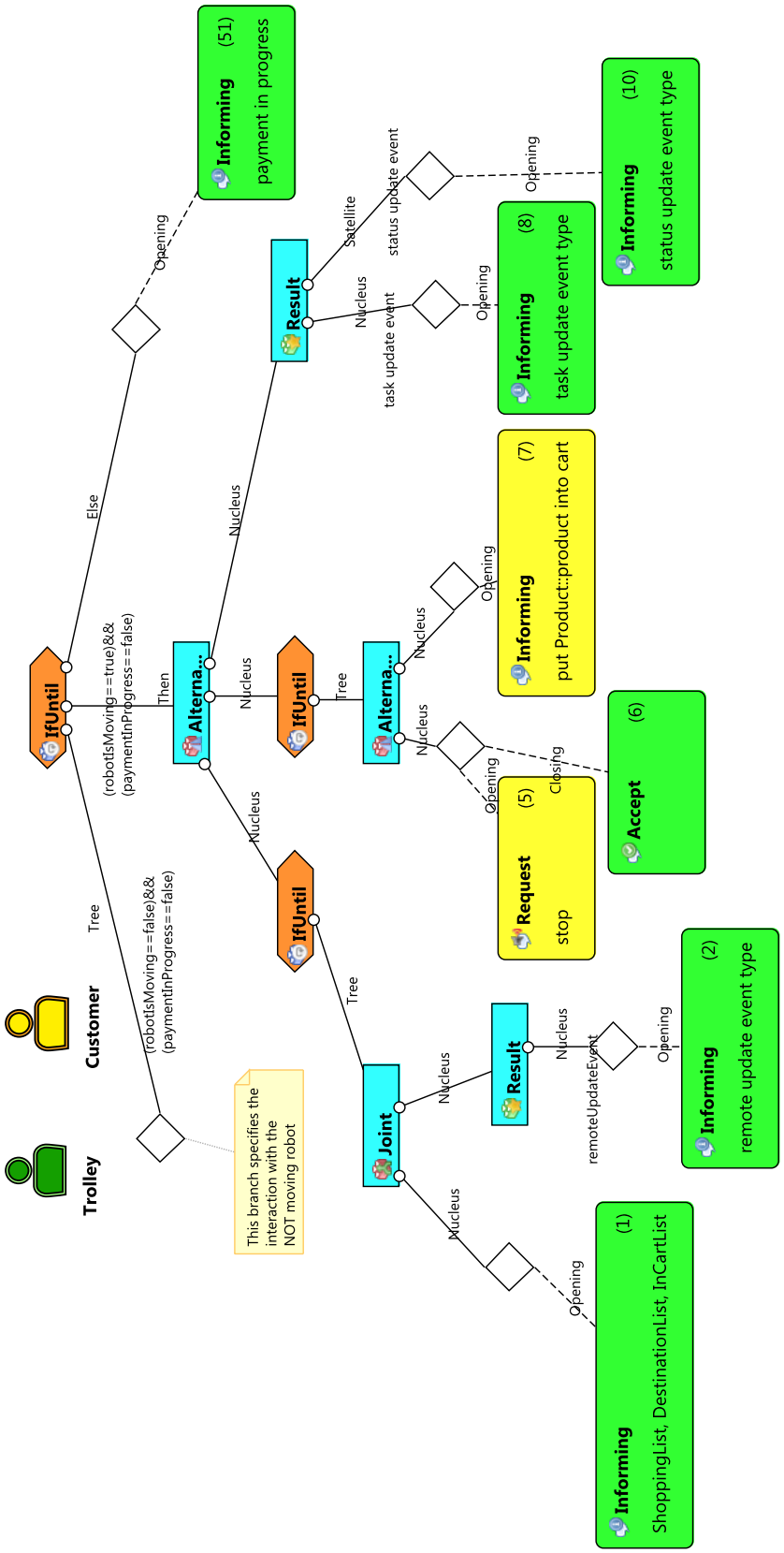


Figure 6.5: Shopping Discourse – Robot is MOVING branch

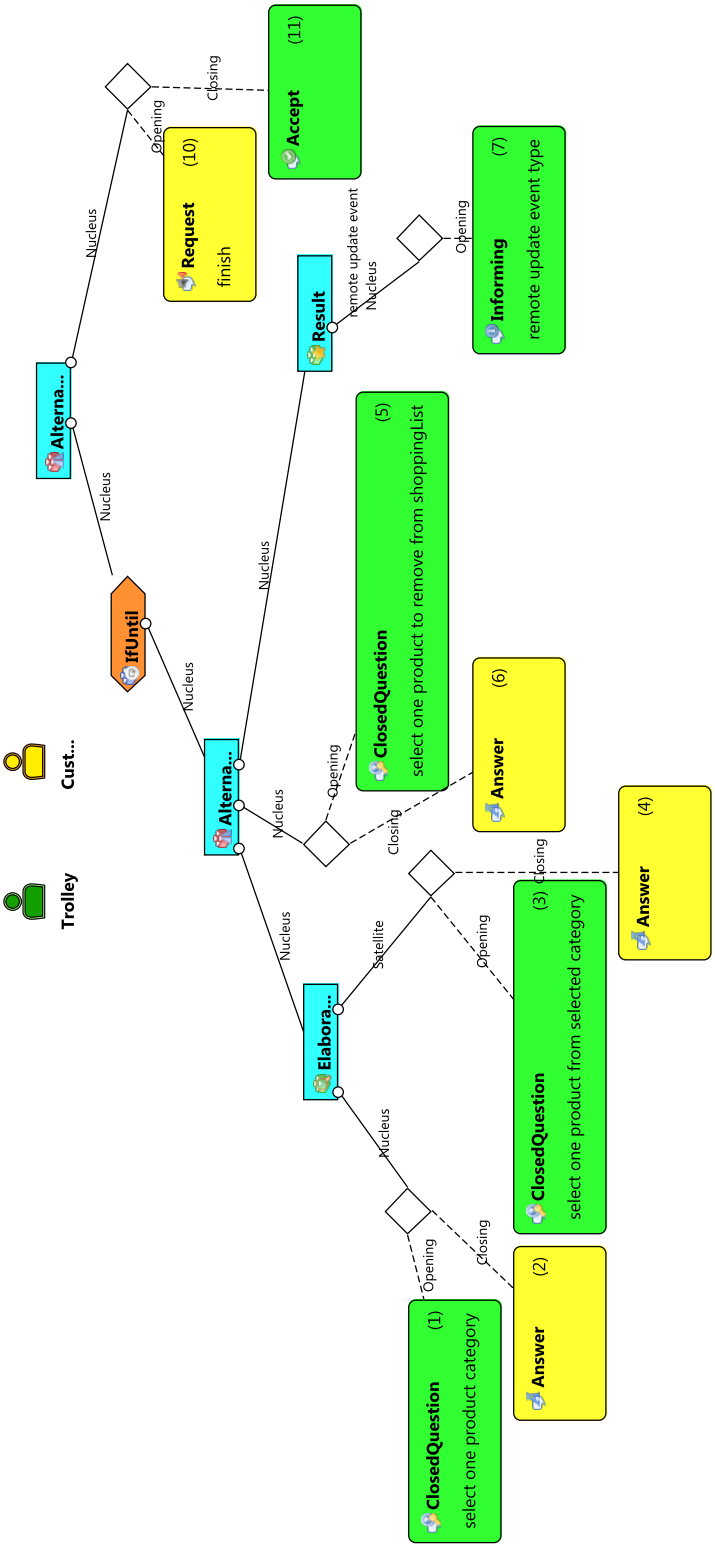


Figure 6.6: Shopping Discourse – Inserted Sequence Manage Shopping List

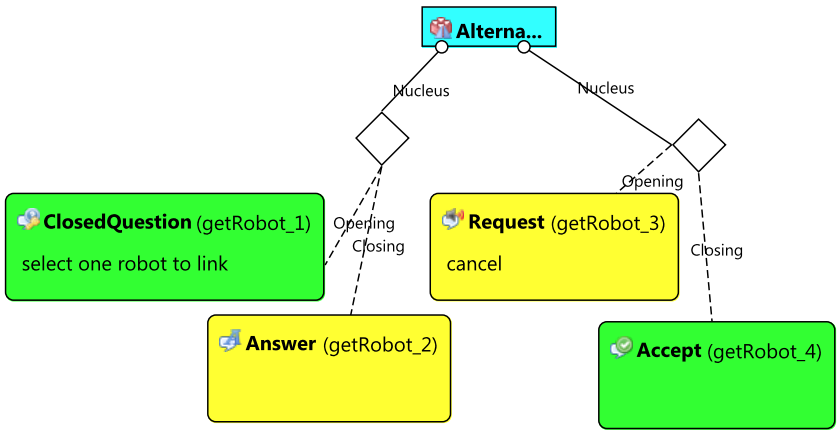


Figure 6.7: Shopping Discourse – Inserted Sequence Link To Other Robot

based WIMP UI generation approach presented in Chapter 3 had to be configured specifically to achieve the given usability and design requirements.

During the design process of transformation rules, which are supposed to create part of the given finger-based touchscreen GUI design, it sometimes happened that the wrong transformation rule was applied. This was caused by a preexisting rule for finger-based touchscreen having exactly the same source pattern but creating a different target pattern (not the given finger-based touchscreen GUI design). In this case the designer has two possibilities to enforce the selection of the desired rule that creates part of the given finger-based touchscreen GUI design. The first possibility is to increase the priority of the desired rule. The second is to make the rule more specific by matching the content. We decided to make the rule discourse specific by matching the content because the given UI design was CommRob specific.

Transformation rule development was a difficult and error-prone task to do. After creating a transformation rule it had to be executed to detect possible errors. Some errors could be detected during the transformation process but others only showed up during runtime. Such a runtime error often occurred when a widget in the target pattern of the rule did not trace to the correct Communicative Act in the source pattern. Thus, during runtime the content could not be displayed. Achieving the state of correctly specified transformation rules was very time consuming.

In CommRob, transformation rules did not always have to be created from scratch. Many transformation rules were of quite similar nature. Therefore, in many cases it was reasonable to create a copy of an existing rule and to start editing it until the desired rule was achieved. This helped to save a lot of time and effort.

The design of transformation rules in CommRob requires imagination to picture the resulting UI part. The target pattern was specified in a tree editor. Thus, the only way to estimate the impact of a newly designed rule was to execute the transformation process and start the CommRob GUI. The process of editing a rule and verifying the result is time intensive.

In CommRob the GUI generation process had to deal with preexisting requirements on what the GUI should look like to achieve reasonable usability. Thus, it had to be defined explicitly how certain pieces of information should be rendered and the layout of some parts of the GUI had to be influenced. As the UI of the CommRob Trolley was multi-modal there were some parts

of the interaction model that were not relevant for the GUI modality at all, so those parts were prevented from being rendered for this finger-based touchscreen GUI [EFK10b, EKKF10].

CommRob-specific rules were introduced to achieve these goals. These rules either match explicitly content objects from the CommRob Domain-of-Discourse model (content-rendering rules), or constellations from the CommRob specific Discourse Model (layout rules and rules that prevent some parts from being rendered).

6.2.1 Adapting Layout According to a Given Design

This section illustrates how customized transformation rules and automatic layouting [Lei10] have been combined to achieve the required GUI layout.

The UI generation process with transformation rules allows the layouting of the UI only by specifying the layout separately for the target pattern of each rule. This means the UI can only be layouted in subparts. During the transformation process, these subparts are connected according to the elements of the Discourse Model. All layout data that is still missing after the Structural UI Model has been completed, is calculated automatically by a layout module [Lei10]. In most cases even the layouting of the subparts is hard to achieve because some of the target pattern elements are replaced by content rendering rules in a second step. Another reason is that n-nucleus relations can have n branches. The number of branches cannot be considered in the layouting of the target pattern of the transformation rule because it is not known during the design time of the generic transformation rules. Therefore, specific CommRob layouting rules had to be specified to satisfy the predesigned GUI layouting requirements. They are tailored to the CommRob Shopping Discourse Model and therefore, only can be created after the Discourse Model has been completed.

The key idea of the CommRob layouting rules can be explained with the following example:

Figure 6.8 illustrates an excerpt of the CommRob shopping discourse. It shows an Alternative relation with four nucleus branches. Two branches are each composed of a Request-Accept adjacency pair, one of a Condition relation and one of a Sequence relation. This discourse subtree is matched by the CommRob layouting rule illustrated in Figure 6.9. The source pattern of the rule is equivalent to the discourse tree in Figure 6.8. It is illustrated in the transformation rule embedded in the element “Discourse”. The adjacency pairs in the rule match the exact name of the adjacency pair in the CommRob Discourse Model. The Target pattern is embedded in the “Panel” element “Alternative1” and is composed of 4 Panels. Each Panel serves as a container for the transformation result of a specific nucleus branch of the Alternative relation. The Mapping element in the rule is used to assign a specific nucleus branch to a specific panel. For example, the left-most nucleus branch of the Alternative relation in Figure 6.8 is assigned to the panel “ManageShoppingList”.

Figure 6.4 illustrates the generated part of the CommRob touchscreen GUI with customized layout, which is embedded in the manually created part of the GUI. One of the rules needed to achieve this layout is the CommRob layouting rule in Figure 6.9. It arranges the Manage Shopping List button in Panel 1, the Continue button in Panel 2, Return Trolley button in Panel 3 and the rest in Panel 4, illustrated with the numbers in the rounded boxes in Figure 6.4. The Manage ShoppingList button on the left, the Continue button in the middle and the Return trolley button on the right of the upper row. In the second row the rest of the UI is placed.

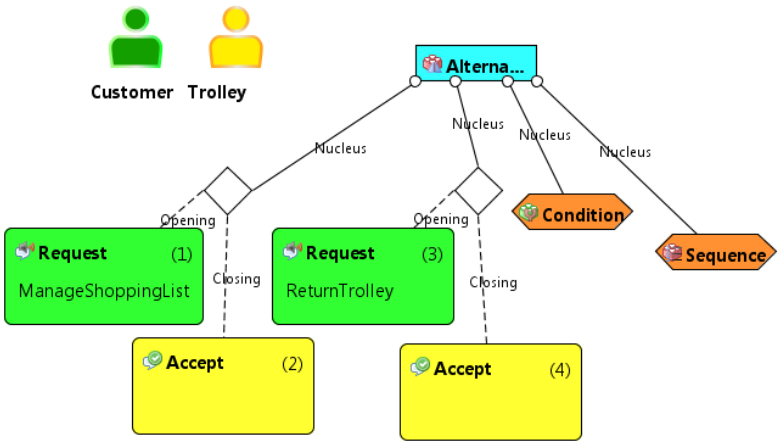


Figure 6.8: CommRob Shopping Discourse Extract

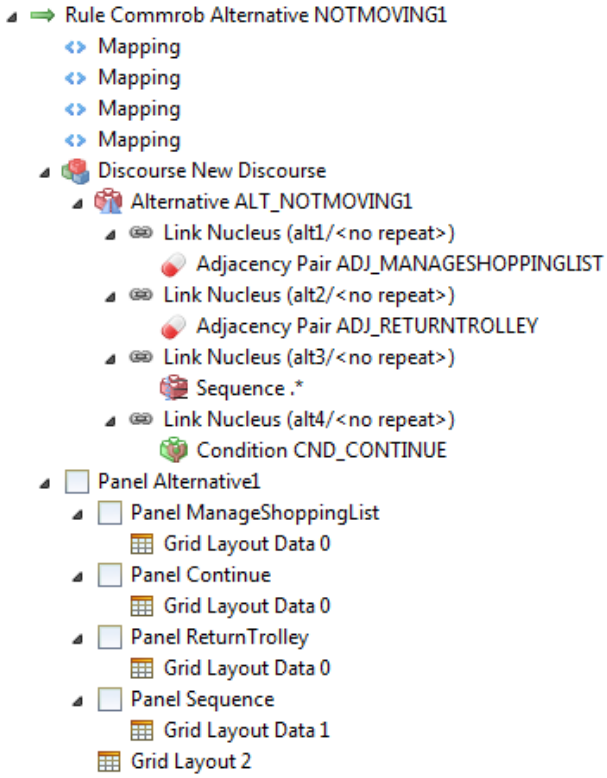


Figure 6.9: CommRob Layout Rendering Rule

In contrast to Figure 6.4, the fully automatically layouted CommRob UI would look like in Figure 6.10. For this generation no CommRob-specific layouting rules were used. The automatic layouting results in the placement of the Return Trolley button below the Manage Shopping List button instead of being placed right next to the Continue button. This is due to the consideration of the predefined screen proportion of 2:3. The automatic layouting can only take information from the Discourse Model into account, like the relation type. The Alternative relation conveys no layout information and, therefore, only the size of the generated Structural UI parts can be considered for layouting.

6.2.2 Presenting Content According to a Given Design

The CommRob content-rendering rules are designed to present content from the CommRob Domain of Discourse Model suitable for a finger-based touchscreen in a predefined way, for example, in a specific order of attributes or different states as icons. Figure 6.11 illustrates the CommRob content-rendering rule for the shopping list. It matches the Closed Question – Answer adjacency pair with a ShoppingList as content of the CommRob Discourse Model in Figure 6.3. It creates a panel composed of three labels and a ListWidget. The ListWidget itself is composed of two picture boxes, one to illustrate the picture and one to illustrate the state and a label to represent the name.

Figure 6.12 illustrates the resulting UI part of the CommRob discourse after applying the rule in Figure 6.11. A list entry is represented by a row with three columns. The left column is occupied with the picture of the product, the middle column is filled with the name of the product and the right column is the icon for the state.

6.2.3 Customising Style According to a Given Design

The style of each widget of a generated GUI can be defined in a Cascading Style Sheet (CSS). This mechanism was used to define a sound for each click on a button or a list. This gives additional feedback to the user about his action. Moreover, the CSS is used to set icons either for widgets (e.g., a button) or to represent enumeration values. Icons on buttons help the user to associate a meaning and to distinguish them more easily. Thus, they help to ease the use of the GUI and to improve its usability.

The mapping from enumeration values to icons is used to represent the status of a product. With one look the user can tell whether the product is in his cart, if he went there but did not take it, or if he still has to go there. Different colors were used for the user's Trolley and the partner Trolley. Thus it becomes easy for the user to see which products are in which Trolley.

6.3 Lessons Learned

During the application of the approach presented in Chapter 3 in the CommRob project, several issues showed up. In this section we generalize from the specific case at hand during the generation of the GUI for a given finger-based touchscreen GUI design.

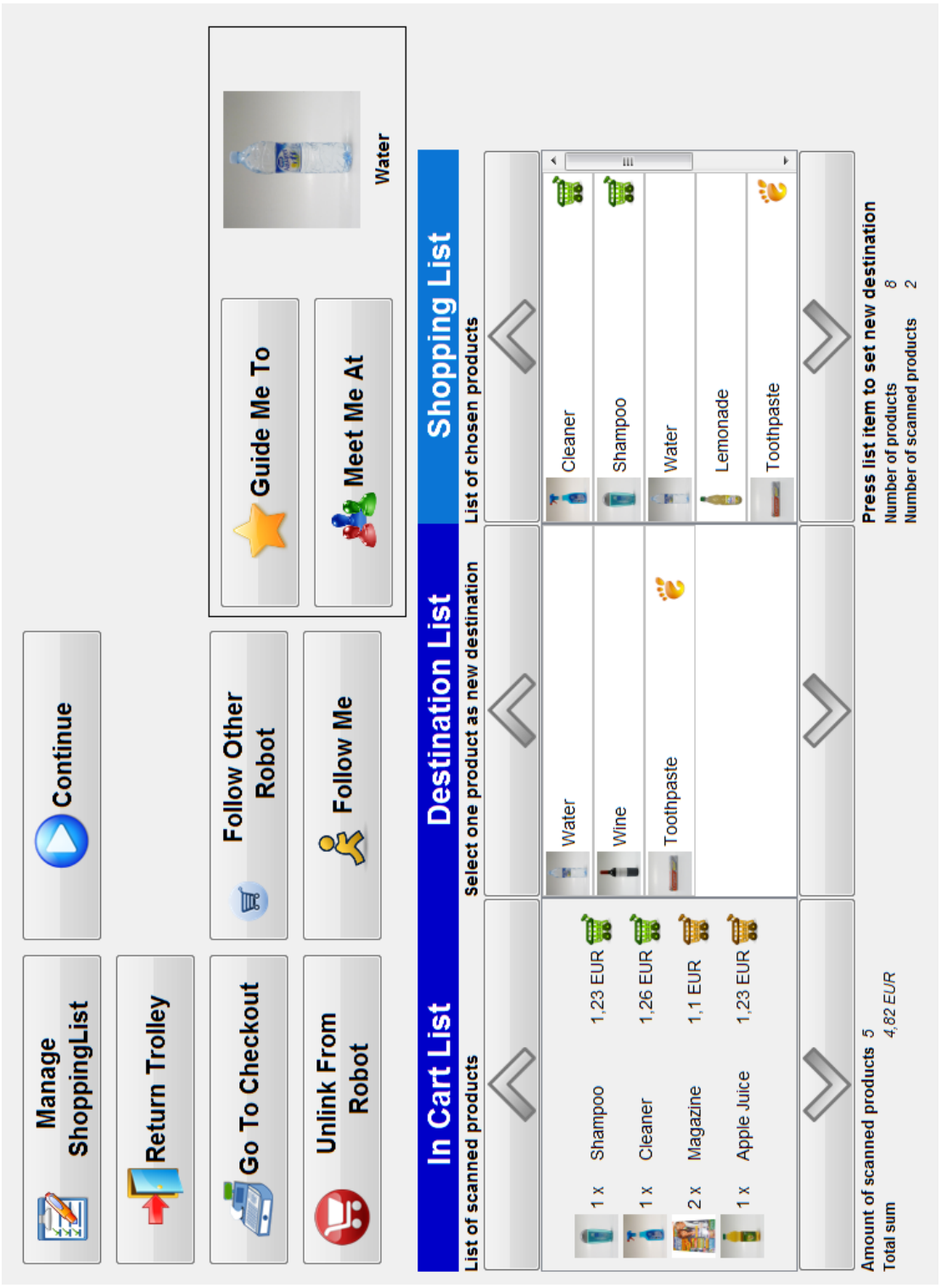


Figure 6.10: CommRob screen with fully automatic layout

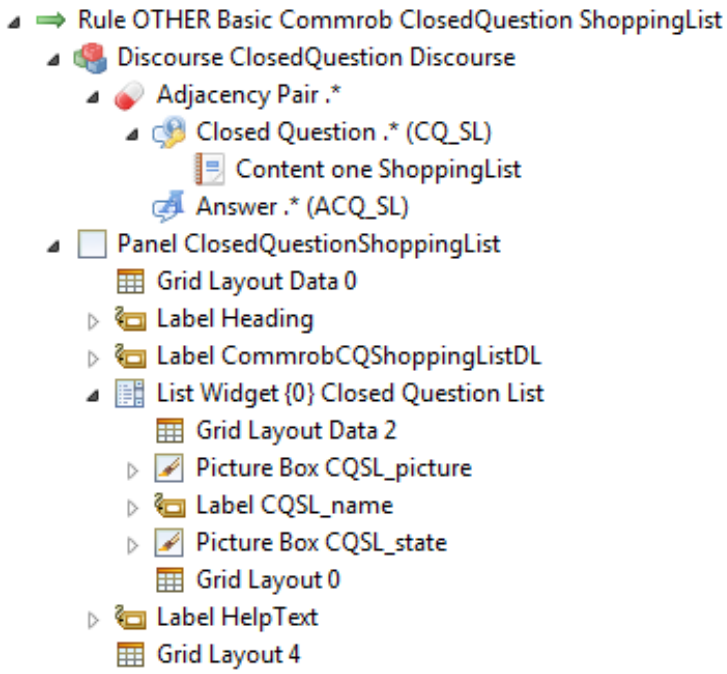


Figure 6.11: CommRob Content-Rendering Rule for ShoppingList

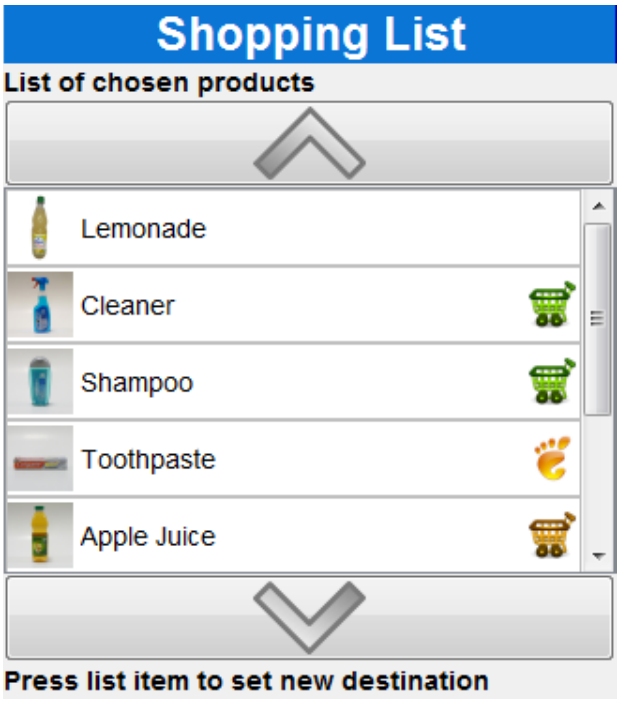


Figure 6.12: CommRob ShoppingList

- **This approach allows the development of GUIs based on a given GUI design:** During the CommRob project, a concrete finger-based touchscreen GUI design was predetermined. Therefore, the UI generation process presented in Chapter 3 had to specifically take care of the UI by the definition of new specific transformation rules. Those were used to adapt the layout according to a given design, to present the content according to a given design and to prohibit the transformation of certain discourse elements. However, the GUI generation for a given GUI design meant a lot of additional effort and is, therefore, better suited for a semi-automatic approach which allows to modify the generation of the GUI more easily.

- **A given GUI design can be achieved with specific transformation rules:** During the CommRob project, it turned out that the transformation rules developed so far where not suitable for the Commrob GUI. The Commrob UI was used with a touchscreen which had to be used by finger and, therefore, needed to take coarse pointing granularity into account. Additionally to the transformation rules for desktop (fine pointing granularity) transformation rules for finger-based touchscreen (coarse pointing granularity) had to be developed to meet the given GUI design. Specific transformation rules had to be developed to adapt the layout according to the given design.

- **Letting the designer choose which rule is applied avoids rule mismatches:** In many cases, there exist different UI constructs for the same discourse element which cannot be distinguished automatically by the tool. Therefore to avoid the rule mismatch that happened in CommRob, the only way to achieve satisfying results for the end user is to let her decide which of the possible constructs she wants to have. This means that during the GUI generation process the designer should be able to choose which of the matching rules she wants to apply. Therefore, the target pattern of each matching rule should be visualized graphically so the designer can choose the appealing one.

- **Automatic layouting is not enough (the possibility to customize the layout efficiently is needed):** This UI generation process allows the layouting of the target pattern of each rule as well as automatic layouting that places the resulting panels according to the available space. If the UI design defines the position of a panel in a different way than the automatic layouting is able to produce, then discourse-specific layouting rules as used in CommRob are needed to place a panel to a specific position on the screen. More efficiently, instead of specifying discourse-specific layouting rules, the layout can be customized to the needs of the user with a graphical Screen Model [Ran10] editor.

- **The GUI transformation process should be modality aware (not-to-render rules):** During the CommRob project the designed discourse models also included communicative acts which were not suitable to be rendered for the GUI modality. Therefore, specific transformation rules had to be designed to prohibit these communicative acts to be transformed by the GUI generation process. Those rules would not be necessary anymore if the GUI transformation process was modality aware. The transformation process should only consider those communicative acts which have been tagged as supported for the GUI modality [EKKF10].

- **Rule metamodel compliance check would help to develop rules more efficiently:** During the CommRob project, a lot of transformation rules had to be developed. Creating transformation rules manually has a high potential for errors. For example, if a widget in the target pattern does not trace to the correct communicative act in the source pattern

then the content cannot be displayed. Such errors might be detected after executing the transformation process but in the worst case only during runtime. A rule metamodel compliance check framework [Sch10] that checks the rules for such errors constantly during the development makes rule design more efficient.

- **Creating new rules based on existing ones by rule specialization:** Many transformation rules used in the CommRob project are similar. Having to copy and edit a rule each time a new rule with a slightly different source or target pattern is desired is difficult to maintain because the slightly difference between rules is hard to find. Rule specialization allows the creation of new rules based on existing ones by only additionally specifying the difference. In contrast to copy and edit it lets the designer focus on the difference between the rules.
- **Visual representation of the rendering rules would help the designer to customize the GUI:** When designing new transformation rules during the CommRob project the designer still needed imagination to picture the resulting UI part, because there was no visual representation of the target pattern of the transformation rule. Therefore, a visual representation of the target pattern of the transformation rule would help the designer to predict the impact in the resulting GUI and enable him to make better rule design decisions.

7 Related Work

This chapter gives an overview of the state of the art related to this work. It presents the existing approaches in the field of semi-automatic GUI generation. The most promising approaches are explained in more detail. First we explain the UI generation with the OlivaNova model execution system of Oscar Pastor [PEPA08]. Second we explain the generation of UIs based on task models of Fabio Paterno [MPS04]. Third we present the UsiXML-based MDA-compliant Environment for developing UIs of Jean Vanderdonckt [Van05]. Finally, other related approaches are covered.

7.1 User interface Generation with the OlivaNova Model Execution System

This section explains the user interface generation approach with the OlivaNova model execution system. Figure 7.1 illustrates it compared to the models in the MDA approach. The MDA approach proposes the definition of the system with an abstract model called CIM illustrated in the top left of Figure 7.1. It is located on the same abstraction level as the Functional Requirements and Interaction Requirements Model of the OO-Method [PEPA08]. The Functional Requirements Model is composed of four models. These are the Mission Statement, the Function Refinement Tree, the Use-Case Model and the Sequence Diagram. As illustrated, the Sequence Diagram corresponds to the PIM of the MDA. The Interaction Requirements Model is composed of two models, the UI Sketches and the Concur Task Tree (CTT) Model. The UI Sketches correspond, as illustrated in the diagram, to the PIM. The dashed line in Figure 7.1 shows that the Sequence Diagram and the UI Sketches as well as the Conceptual Model correspond to the PIM. Following the MDA approach, the CIM is transformed, into a more concrete model, the PIM. It corresponds to the Conceptual Model of the OO-Method, which is composed of the Object Model, the Dynamic Model, the Functional Model and the Presentation Model. The Object Model represents the object structure and its static interactions. The Dynamic Model defines the interactions between the objects. The Functional Model specifies how events change the object states. The Presentation Model represents the interaction between the system and the user. As already mentioned, the UI Sketches and the Sequence Diagram of the OO-Method correspond to the PIM of the MDA. The next step in the MDA approach transforms the PIM to the PSM, which is specific to a certain platform. It corresponds to the Compilation Model of the OO-Method. The last step in the MDA approach transforms the PSM to the Code Model. It corresponds to the Source Code Model in the OO-Method, which is composed of three layers: the Interface Layer, the Business Layer and the Persistence Layer.

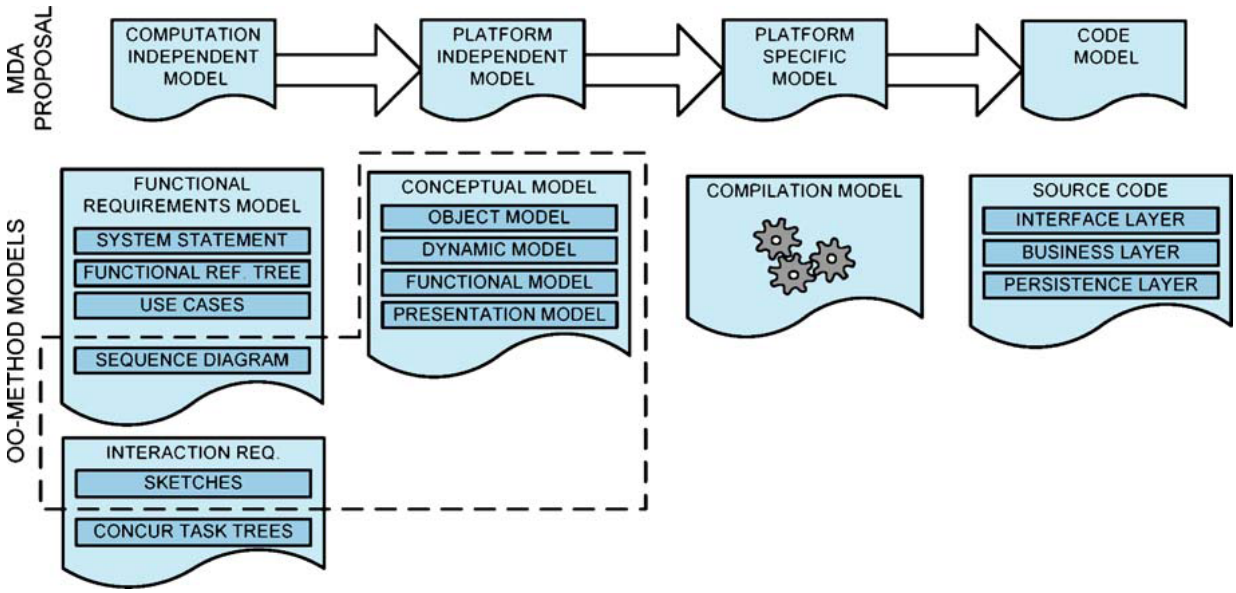


Figure 7.1: The software generation process of the OO-Method (copied from [PEPA08])

Figure 7.2 illustrates the derivation process in the OO-Method. Derivation means the creation plus the automatic mapping (transformation) of one model to another one. The process starts either with the specification of the Sequence Diagram or the definition of Use Case Templates. Then the UI Sketches are created manually according to the previously defined Use Case Templates or Sequence Diagrams. The corresponding Task Tree is automatically generated. The Object Model can be derived either from the Sequence Diagram or the Use Case Templates. The Presentation Model is derived form the Task Tree [EPP06]. Then the Object Model and the Presentation Model are manually related to each other. However, it is planned to achieve this mapping automatically. The Dynamic Model and the Functional Model have to be created manually according to the Functional Requirements Model. When all the models of the Conceptual Model have been specified, the application is generated automatically by a model compiler.

The models used in the approach presented in Chapter 3 of this work can also be mapped to the MDA models. The Discourse Model corresponds to the CIM and is directly transformed to the Structural UI model, which corresponds to the PSM. There is no PIM in this approach. The Code Model corresponds to the source code generated out of the Structural UI for a particular target platform.

7.2 Generation of UIs based on Task Models

Figure 7.3 illustrates the UI generation approach based on Task Models. It starts with the specification of a task model which is platform independent. The Task Model has to be manually annotated for each platform to a System Task Model which is platform specific. For each System task model an Abstract UI model is generated which is also platform specific. Subsequently, the Abstract UI model for each platform is transformed to a Concrete UI model which is platform specific as well. Finally the Concrete UI model is used to generate the UI source code for the particular platform. For more detailed information about this approach see [MPS04].

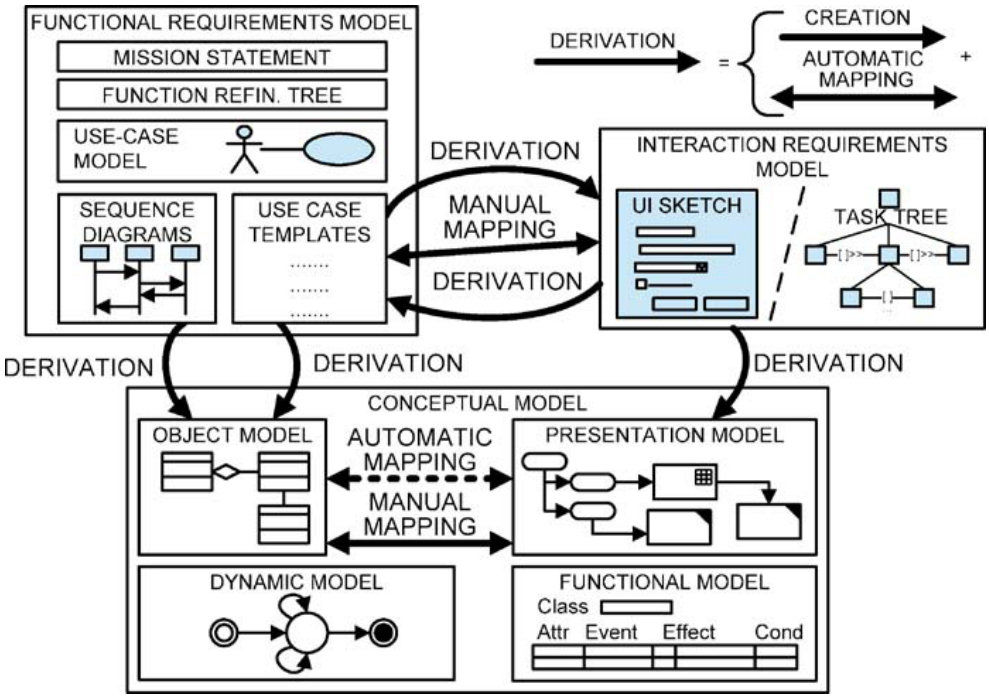


Figure 7.2: OOMethod derivation (copied from [PEPA08])

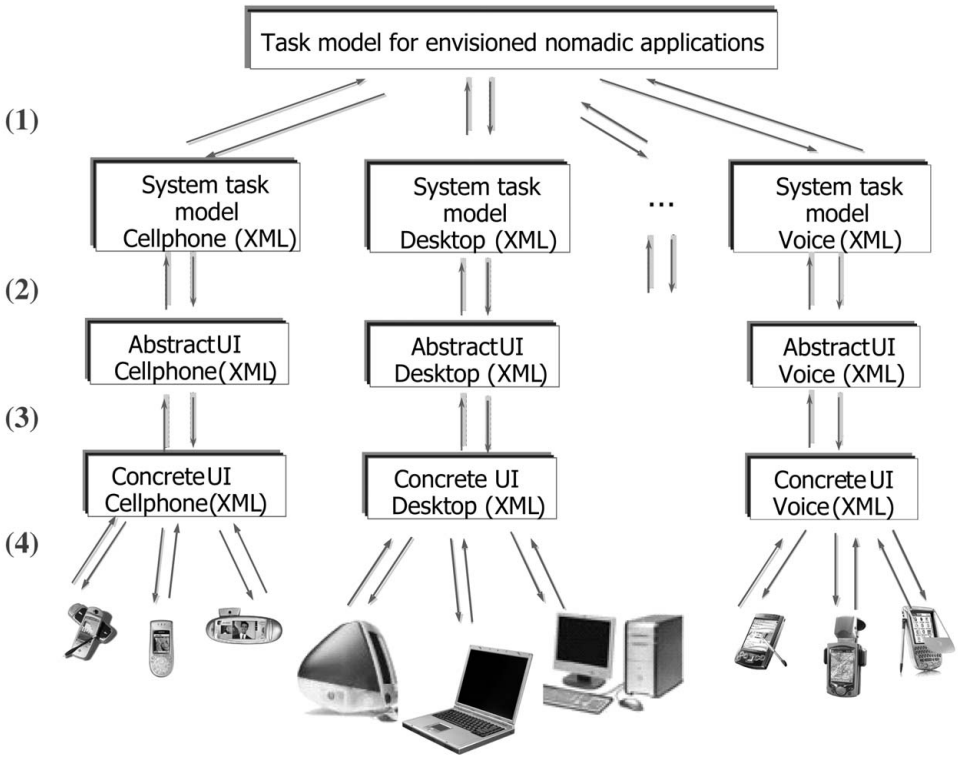


Figure 7.3: One model many interfaces approach based on task models (copied from [MPS04])

An advanced approach to specifying multi-device user interfaces based on Task Models instead of Discourse Models is presented in [MPS04]. Its basic approach is to start modeling tasks and to generate user interfaces for diverse devices according to specific device characteristics. In contrast to the discourse-based UI generation approach, some of the transformations between models are done semi-automatically or manually. The transformations are implicitly coded in the system, there is no genuine transformation engine like ATL, ATOMS3 or MOLA. A major difference between Task Models and Discourse Models is that task models express only temporal dependencies whereas discourse models specify causal dependencies, too. The semantic mapping of task models to dialog models based on UML State Machines is explained in [dBC07]. A similar mapping from Discourse Models to UML State Machines in the course of behavior generation is presented in [PFA+09]. Additionally an algorithm is presented to derive the presentation units of a UI. Furthermore, instead of tasks the approach presented in Chapter 3 of this work understands user interfaces as communication with intentions between a user and a machine.

Paterno et al. [PSS09] describe a method and a model-based language called MARIA for creating interactive applications based on pre-existing functionalities in multi-device contexts. It supports the generation of UIs for accessing multiple services. It allows both the specification of service-to-service interaction as well as the approach presented in Chapter 3 of this work. The MARIA model has to be extended for UI generation with annotations, whereas the discourse-based GUI generation approach does not need any modifications for UI generation. The discourse models are capable of expressing service-to-service communication as well as serving as a UI generation starting point per se. Discourse models support the explicit expression of the intention of a communication by the use of communicative acts.

The differences between this approach and the discourse-based UI generation from Chapter 3 are:

- **Platform-specific starting point for UI generation:** To enable the UI generation the platform-independent Task model has to be refined to a platform-specific System task model. This approach starts the UI generation from a platform-specific model (System task model) which has a lower abstraction level than the starting point of the approach presented in Chapter 3 of this work. In contrast, the approach presented in Chapter 3 of this dissertation starts the UI generation from a CIM, the Discourse Model, which is platform independent.

- **Approach has no MDA compliance:** As illustrated in Figure 7.3, all models involved in the generation process are platform-specific. The Abstract UI model is according to the MDA a platform-independent model, however in this approach it is platform-specific. In contrast, the discourse model based UI generation approach is MDA compliant. However, the UI generation skips the PIM and directly transforms the CIM to the PSM.

- **Tasks can have only temporal relations:** The CTT notation only allows the connection of tasks in a temporal manner. Semantical relations that might exist are not captured in the model. Therefore, this semantical information is not available during the GUI generation process. In comparison, Discourse Models also relate adjacency pairs according to their semantic relationship. This enables the discourse-based UI generation process to layout the GUI according to the semantic relation. The Background relation, for example, which is explained in Chapter 3, expresses that the satellite-branch conveys background information related to the nucleus-branch. In this case the nucleus-branch is rendered on the right and the satellite-branch on the left).

In contrast to this approach, the approach presented in Chapter 4 of this doctoral dissertation does not start from high-level task models, but from detailed requirements specifications. Alternatively, it can start from concrete UIs and lead automatically to artifacts in a requirement's specification.

7.3 UsiXML-based MDA-compliant Environment for Developing UIs

This section gives an overview of the UsiXML-based MDA-compliant Environment for developing UIs. UsiXML is a User Interface Description Language (UIDL), which is used to specify UI models. Figure 7.4 illustrates the UsiXML-based UI generation process as well as the supporting tools. Starting with the specification of UsiXML-based task and domain models, graph transformations are used to generate an Abstract User Interface in UsiXML. The next step transforms the AUI into a Concrete User Interface in UsiXML also by the use of graph transformations. The reason why graph transformations have to be used is, that the transformation metamodel incorporates classes which are strongly related to the structure of graphs. In the final step, the CUI is rendered to a FUI. In this approach all the models are specified in UsiXML, even the graph transformations. In contrast, the approach presented in Chapter 3 of this work uses a different language for each model.

The most important UsiXML tools supporting this UI generation approach are:

- TransformiXML [LV04b, LV04a]: TransformiXML is a tool for the application of model-to-model transformations. A UsiXML compliant specification is transformed by the application of transformation rules into another UsiXML compliant specification. Transformations are possible between the three top development stages of the Cameleon Reference Framework to support forward and reverse engineering. The transformation rules are expressed in UsiXML. The theoretical background to execute and express transformations is based on graph grammars.
- IdealXML [MLJ06]: IdealXML provides a graphical editor for the task model, the domain model, and the AUI model. It also allows the transformation between this models by calling TransformiXML. It is a pattern-based environment which allows to store, edit and manipulate interaction patterns and to reuse them in UI development.
- KnowiXML [FFS+04]: KnowiXML is a Knowledge-based System (KBS) that facilitates during the generation of AUIs the application of models and the allocation of appropriate visual elements. The theoretical background for generating the AUIs is based on problem solving strategies from Artificial Intelligence (AI). The knowledge of the interface designer is encoded in KnowiXML and modifies UI specifications through the use of a User Interface Description Language (UIDL).
- GrafiXML[MV08]: GrafiXML is a CUI editor that enables designers to build multi-target UIs based on UsiXML. It maintains model consistency between three representations (internal: UsiXML specification, external: interface preview, conceptual: UI model) through a set of mappings based on a UI ontology. Thus, GrafiXML provides a unique set of features for supporting designing interfaces for multiple targets. It supports 5 levels of independence (device, platform, channel, modality, and context of use) in expressing the CUI. It also supports the automatic generation of UI code in Java, XHTML, HTML and XUL.

- SketchiXML [CVL06]: SketchiXML is an editor that enables designers, developers, or even end users to sketch UIs with different levels of details and support for different context of use. The sketching is analyzed to derive a CUI model. UI elements of the sketch which are not recognized are saved as images. Another CUI editor, e.g., GrafiXML can then be used to refine the specification and automatically generate the UI code.
- VisualiXML[SV04]: VisualiXML is a tool to generate a UI from a CUI model based on generative programming.

The UsiXML tools supporting the specification of different models and transformations are illustrated in Figure 7.4. For more details about this approach see [Van05].

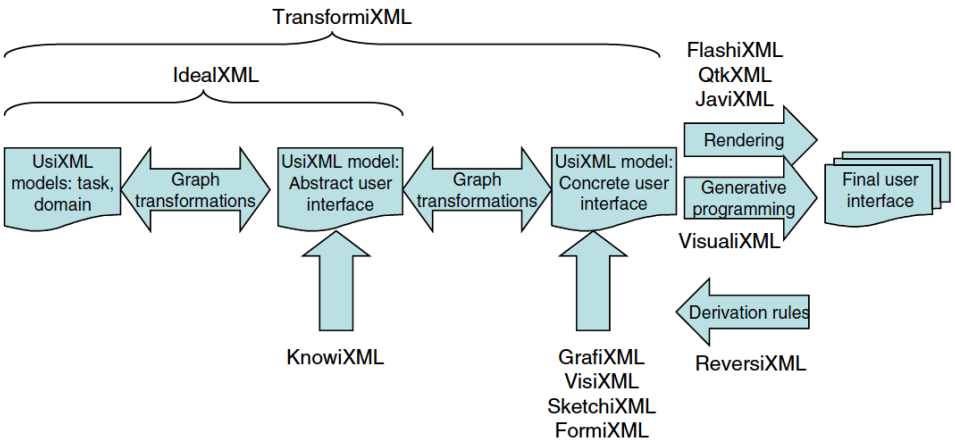


Figure 7.4: UsiXML based UI generation approach and supporting tools (copied from [Van05])

7.4 Other Related Approaches

Model-based UI design methods developed and published in the nineties including OVID [RBIM97], and Idiom [vH94] focus on creating different kinds of models, like user's conceptual models, task models and interaction models. Unlike the approach presented in Chapter 3 of this work, which is *model-driven*, all the mentioned approaches above are *model-based*. That is, they allow expressing an interactive system by task, concept and/or abstract models in a first step and use them in an informal process or in a sequence of systematic steps to construct a user interface. They do not support transformations between models on different abstraction levels.

In contrast, UI Frameworks like XUL[1] (XML User Interface Language) are able to generate UIs automatically but they rely on UI models at the abstract widget level, whereas Discourse Models are located on the task level, which is on a higher level.

Florins *et al.* describe in [FSV+06] transformation rules for pagination of UIs on different levels. Pagination is a way of providing a limit and an offset to the number of items fetched, e.g., from a database and presented to the user in parallel. Partitioning the Discourse Models into presentation units in the first transformation step provides important guidance for pagination [BFK+08].

[1] `http://www.mozilla.org/projects/xul/`

Botterweck shows in [Bot06] a model-driven approach that starts on the abstract UI level and contains rich procedural UI descriptions together with UI elements. Thus, it requires UI modeling as well as dialogue modeling. In contrast the approach presented in Chapter 3 of this doctoral dissertation starts on the task and domain level and thus no UI modeling and dialog modeling is involved.

When a GUI is generated for different devices, a mechanism can be applied that transforms a potentially large interface (highest degree of parallelism) into a more compact (serialized) form. An easy approach is just to add scroll bars to a GUI developed for a PC application which shall be used, e.g., on a PDA with reduced screen size. This potentially results in an annoyed user, as scrolling is often required when using the interface.

To circumvent this drawback, a technique presented in [XLH+09] has been developed which automatically transforms Web pages into hierarchically structured subpages. This approach considers the size of the screen, the size of page blocks, the number of blocks in each transformed page, the depth of the hierarchy that the subpages form and the semantic coherence between the blocks. "Size" is actually interpreted here in terms of number of pixels, i.e., screen resolution. Another approach to taking screen size into account for generating multi-target user interfaces can be found in [CVC08], but it actually interprets size as screen resolution as well. It resizes based on given numbers of pixels on the screen. Note, that such approaches are difficult to apply for touch screens with variable screen resolution, since the metric size of input widgets matters there.

Design guidelines for finger-based touchscreens are described in [Gal02], e.g., that the object separation should be at least $1/8''$. As another example, when the consequences of selection are destructive, i.e., a "remove" or "delete" functionality, confirmation helps to avoid inadvertent selection.

The minimum size for targets on a touch screen is mentioned in [PKB06]. This study suggests a size of 9.6mm with 5% erroneous trials for discrete screen objects. Previous studies such as [PHPC08] investigated touch key design for target selection on a mobile phone, more precisely the size of objects and their location on the screen. However, this study focused on mobile phones, which have more constraints on the screen size than, e.g., touch screens for kiosk applications. A main result of this study was that larger touch key sizes lead to higher performance and more subjective satisfaction. The size of the targets on a touch screen is also part of an empirical study in [Ben99]. It points out that large targets should be used whenever possible, having benefits like reduced contact time and fewer errors. However, a drawback of large targets are fewer manipulable objects on the screen.

The UI Pilot approach by Puerta *et al.*[PMM05] is semi-automatic by requiring the designer to specify tasks and a so-called wireframe, which is a sketch of the proposed behavior, structure, navigation, and content layout of the GUI, for the user interface. Afterwards, the tool can suggest widgets for each user interface element. This approach provides more flexibility to the user interface designer by letting her decide which of the suggested widgets is used. However in contrast the discourse-based GUI generation approach allows fully automatic content presentation.

Elkoutbi *et al.*[EKK06] present an approach that generates a user interface prototype from scenarios. Scenarios are enriched with UI information and are automatically transformed into UML statecharts, which are then used for UI prototype structure and behavior generation. In contrast to this approach, the discourse-based GUI generation approach models classes of dialogues supporting a set of scenarios. The discourse models are transformed to UML statecharts as well, but we do not have to enrich them for this purpose.

Pederiva *et al.*[PVE+07] describe a beautification process that helps a designer to improve a generated user interface via a constrained user interface editor. The manual modifications to make a UI look more beautiful are called beautification. This editor allows applying beautification operations to specific UI elements, resulting in model-to-model transformation. Since the discourse-based GUI generation approach involves content presentation according to purpose, beautification should be less important for this part of UI generation.

A transformation system that fits Web pages automated and on-the-fly to screens of small devices is presented in [XLH+09]. The transformations are performed in order to minimize navigation and scrolling like in the discourse-based GUI generation approach. In contrast, however, this process alters an already existing UI.

Declarative user interface specifications are used as input for multi-target UI generation in [GW04]. The user interface adaption is treated as an optimization problem based on a user- and device-specific cost function. Compared to such user interface specifications, discourse models are on a higher level of abstraction.

The model-driven approach for engineering multi-target UIs presented in [CVC08] supports switching between generated presentations during runtime. The discourse-based UI generation approach, in contrast, is intended to automatically generate GUIs for different screens of small size from a single discourse model.

In the literature, a system that allows the automatic derivation of inverse transformation rules is called Automatic Round trip Engineering System (ARES). A systematic method to construct ARES is explained in [Aßm03]. However, there is no solution given on how the inverse transformation rules can be derived automatically.

Using transformations as a bridge between different "worlds" (requirements and UI specifications) can also be found in the work presented by Panach *et al.*[PEPP08]. They propose a method to bring Software Engineering and Human-Computer Interaction closer together. They capture interactions with sketches and transform them into structural patterns of CTT. Similarly to the UI generation approach based on requirements models as presented in Chapter 4 of this work, they transform up and downwards in the reference framework [CCT+03]. In addition, the UI generation approach based on requirements models as presented in Chapter 4 of this dissertation transforms horizontally (on the same level of abstraction) and between different "worlds". Unlike other approaches like the model-driven prototyping proposed in [MBR07], the approach presented in Chapter 4 of this work also transform between models on the *same* abstraction level.

In [CH03] a classification of model transformation approaches is given. Bidirectional transformations can be achieved using bidirectional rules or by defining two separate complementary unidirectional rules, one for each direction. The UI generation approach based on requirements models presented in Chapter 4 of this work uses two separate unidirectional rules to achieve bidirectional transformations. It is explained that declarative rules can often be applied in the inverse direction, too. However, since different inputs may lead to the same output, the inverse of a rule may not be a function. For procedural rules like in MOLA, there is no approach explained how to derive the inverse systematically.

In [CFH+09], the state of the art in bidirectional transformations is presented. Transforming between different views is most suitable with a bidirectional transformation language because the checking required to ensure consistency of two separate transformations is hard, error prone, and likely to cause inconsistency. However, the approach presented in Chapter 4 of this work shows how to automatically derive inverse transformation rules from an existing set of unidirectional

transformation rules defined in a unidirectional transformation language (MOLA). By automatically generating the inverse transformation rule to a unidirectional one, we assure consistency.

In [Ste07], bidirectional model transformations in QVT are analyzed and some semantic issues are discussed. It is argued that the practical use of bidirectional transformations is hindered because there is no guarantee that a hand-written or automatically derived inverse transformation really is an inverse of a given forward transformation. However, automatically derived inverse transformations can assure consistency and help to reduce the transformation rule specification effort.

8 Conclusion and Future Work

This chapter draws the conclusions and presents some directions for future work.

8.1 Conclusion

In this dissertation, I present a new approach to generating Structural UI Models by applying model-driven transformations to Discourse Models[1]. Discourse Models are derived from results of human communication theories, cognitive science and sociology and are used for specifying interaction design of human-computer interaction of information systems. Thus, they contain additional metainformation, like the intention of an interaction, which allows defining sophisticated transformation rules to transform the Discourse Models to Structural UI Models. The transformation takes already device constraints into account to generate a UI structure well suited for the target device, but the resulting UI models are still independent of UI toolkits.

The usual device properties taken into account for automated GUI generation are extended to allow even application-specific UI generation. Information on pointing granularity for a touch screen is included. In fact, these are device properties beyond those for physical devices. The extended specifications are, therefore called, application-tailored device specifications. Transformation rules for different pointing granularities are defined. Fine pointing granularity leads to smaller input widgets on the screen, whereas coarse pointing granularity results in larger input widgets. Since the CommRob feasibility study needed UIs generated for a finger-based touch screen, specific model-transformation rules for coarse pointing granularity were developed.

Model-transformation rules for fine pointing granularity (e.g. desktop) were also presented. They were used for semi-automatically generating a usual GUI for use with a mouse as well. So, we have been able to generate both GUIs semi-automatically from the same high-level discourse specification. This shows that the transformation rules support the generation of UIs for different pointing granularities. This flexibility in semi-automatic GUI generation is unique. In particular, it is based on application-tailored device specifications.

The problem of the presentation of content from the Domain of Discourse according to its purpose defined in the discourse model is addressed. This purpose relates to the intention indicated by the

[1] As mentioned in Chapter 1 Discourse Models for general human-machine and machine-machine communication have been developed in a team effort in the course of the FIT-IT OntoUCP project (No. 809254/9312, www.ontoucp.org) [FKH+06, BFK+08, BKFP08, PFA+09].

type of the communicative act that refers to propositional content to be presented. This approach takes this type into account in the course of automatic content presentation. In this way, it leads to generated user interfaces with content presentations according to purpose. This shows that the transformation rules support the presentation of content according to purpose.

A UI generation process that allows the same rule set to be used for generating UIs for devices with different resolutions is introduced. Through the automatic calculation of space need, it may even have an advantage in this respect as compared to a human interface designer.

Additionally this doctoral dissertation presents a model-driven approach of forward and inverse transformations between Requirements Specifications, UI Specifications and UI Prototypes. We are not aware of any transformation approach between these different "worlds", which are partly even on the same level of abstraction. And it is new to have transformations back and forth between the same pair of models. This approach involves inverse transformations as well. This dissertation proposes an approach for creating inverse rules semi-automatically through metarules whenever possible.

Explicit transformations may help to bridge the usual gap between separated Requirements Specifications, UI Specifications and UI Prototypes. It may also make the overall development more efficient, since it makes explicit use of artifacts from any one "world" to create artifacts in the other one.

Furthermore we present a comparison between a declarative and a procedural transformation language. The declarative transformation language, specifically DTL, is used in the discourse-based UI generation approach presented in Chapter 3 of this work. The procedural transformation language, particularly MOLA, is used for the UI generation approach based on requirements models presented in Chapter 4 of this dissertation. The advantages of each transformation language are discussed and a summary of the comparison is presented. We observed that writing rules in MOLA is much more work than in DTL, because the designer has to take care of the execution order by herself in MOLA, whereas in DTL it is handled by the framework. Thus MOLA can be more suitable for expert users because it allows them to have more influence on the transformation process, whereas DTL is more end user friendly because it allows the designer to focus on the development of the transformation rules.

Finally a Feasibility Study shows the applicability of the discourse-based UI generation approach. In particular, the GUI for the touchscreen of a robot trolley has been partly generated. It illustrates that high-level models and transformation rules capture all necessary information for WIMP UI structure generation.

8.2 Future Work

There are several areas in which this work can be extended:

Development of a graphical rule editor: Rendering GUI panels and other groups of Communicative Act (e.g., alternative sequences in speech) is currently governed by rules which are based on matching the respective set of Communicative Acts. These rules are currently wired in, and cannot be visualized and changed by the modeler easily. A designer would benefit if she can see a visual representation of the rendering rules that produce a particular panel in the final GUI. Therefore, a graphical rule editor could be developed which would allow easy editing for the creation of transformation rules.

Extending the approach to an interactive UI generation process: The approach presented in Chapter 3 of this dissertation has restrictions when a GUI for a given given UI design should be developed, which has been shown in the feasibility study in Chapter 6. The only place for tweaking are the transformation rules, which can only influence the UI subpart generated by them. Therefore, interactive support for the human designer is needed to enable her to make informed design decisions, thus leading to a more satisfying result for the end user as well as making the UI development for a given UI design much easier for the designer. More details about this planned extension of this approach can be found in [Ran10].

Transform UI Sketches automatically to Discourse Models: Discourse Models might be intuitive to many people but if it comes to the GUI generation they might be to abstract for many end users. Therefore, allowing end users to sketch UIs by hand and let them automatically be transformed to Discourse Models could even simplify the UI development process for end users. Other approaches like [KCV10],[PEPA08] are already trying to transform UI sketches to Concur Task Trees. The existing knowledge could be used to support the development of transformations from UI Sketches to Discourse Models.

Literature

[Aßm03] Uwe Aßmann. Automatic roundtrip engineering. *Electronic Notes in Theoretical Computer Science*, 82(5):33 – 41, 2003. SC 2003, Workshop on Software Composition (Satellite Event for ETAPS 2003). 88

[BEF+10] C. Bogdan, D. Ertl, J. Falb, A. Green, S. Kavaldjian, D. Raneburger, and A. Szep. D7.4 report on development of dialogue design support features. Technical report, EU Project CommRob, 2010. 65

[Ben99] Gregory T. Bender. *Touch Screen Performance as a Function of the Duration of Auditory Feedback and Target Size*. PhD thesis, Wichita State University, 1999. 87

[BFK+08] Cristian Bogdan, Jürgen Falb, Hermann Kaindl, Sevan Kavaldjian, Roman Popp, Helmut Horacek, Edin Arnautovic, and Alexander Szep. Generating an abstract user interface from a discourse model inspired by human communication. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS-41)*, Piscataway, NJ, USA, January 2008. IEEE Computer Society Press. 1, 13, 86, 90

[BKFP08] Christian Bogdan, Hermann Kaindl, Jürgen Falb, and Roman Popp. Modeling of interaction design by end users through discourse modeling. In *Proceedings of the 2008 ACM International Conference on Intelligent User Interfaces (IUI 2008)*, Maspalomas, Gran Canaria, Spain, 2008. ACM Press: New York, NY. 1, 13, 90

[Bot06] Goetz Botterweck. A model-driven approach to the engineering of multiple user interfaces. In *Proceedings of the MoDELS'06 Workshop on Model Driven Development of Advanced User Interfaces*, Genova, Italy, Oct. 2006. CEUR-WS. 87

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289 – 308, 2003. Computer-Aided Design of User Interface. 8, 45, 88

[CFH+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *ICMT*, pages 260–283, 2009. 88

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003. 88

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006. VII, 6, 54

[CL99] Larry L. Constantine and Lucy A. D. Lockwood. *Software for use: a practical guide to the models and methods of usage-centered design.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999. 46

[CVC08] Benoît Collignon, Jean Vanderdonckt, and Gaëlle Calvary. Model-driven engineering of multi-target plastic user interfaces. In *Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems (ICAS 2008)*, pages 7–14, Washington, DC, USA, 2008. IEEE Computer Society. 87, 88

[CVL06] Adrien Coyette, Jean Vanderdonckt, and Quentin Limbourg. Sketchixml: A design tool for informal user interface rapid prototyping. In *RISE*, pages 160–176, 2006. 86

[dBC07] Jan Van den Bergh and Karin Coninx. From task to dialog model in the UML. In *Proceedings of the 6th International Workshop on Task Models and Diagrams for User Interface Design (TAMODIA 2007), LNCS 4849*, pages 98–111, Toulouse, France, Nov 2007. Springer. 84

[EFK+10a] D. Ertl, J. Falb, H. Kaindl, S. Kavaldjian, R. Popp, D. Raneburger, and A. Szep. D5.5 final report on the development of the communication platform. Technical report, EU Project CommRob, 2010. 65

[EFK10b] Dominik Ertl, Jürgen Falb, and Hermann Kaindl. Semi-automatically configured fission for multimodal user interfaces. In *ACHI*, pages 85–90, 2010. 74

[EKA+11] Dominik Ertl, Hermann Kaindl, Edin Arnautovic, Jürgen Falb, and Roman Popp. Discourse-based interaction models for recommendation processes. In *The Fourth International Conference on Advances in Computer-Human Interactions, ACHI 2011*, 2011. 66

[EKK06] Mohammed Elkoutbi, Ismaïl Khriss, and Rudolf K. Keller. Automated prototyping of user interfaces based on UML scenarios. *Automated Software Engineering*, 13(1):5–40, 2006. 87

[EKKF10] Dominik Ertl, Sevan Kavaldjian, Hermann Kaindl, and Jüergen Falb. Semi-automatically generated high-level fusion for multimodal user interfaces. In *Proceedings of the 43nd Annual Hawaii International Conference on System Sciences (HICSS-43)*, Piscataway, NJ, USA, 2010. IEEE Computer Society Press. 74, 79

[EPP06] Sergio España, Inés Pederiva, and José Ignacio Panach. Integrating model-based and task-based approaches to user interface generation. In *CADUI*, pages 253–260, 2006. 82

[FFS+04] Elizabeth Furtado, Vasco Furtado, Kênia Soares Sousa, Jean Vanderdonckt, and Quentin Limbourg. Knowixml: a knowledge-based system generating multiple abstract user interfaces in usixml. In *TAMODIA*, pages 121–128, 2004. 85

[FKH+06] Jürgen Falb, Hermann Kaindl, Helmut Horacek, Cristian Bogdan, Roman Popp, and Edin Arnautovic. A discourse model for interaction design based on theories of human communication. In *Extended Abstracts on Human Factors in Computing Systems (CHI '06)*, pages 754–759. ACM Press: New York, NY, 2006. 1, 13, 90

[FSV+06] Murielle Florins, Francisco Montero Simarro, Jean Vanderdonckt, Benjamin Michotte, and Benjamin Michotto. Splitting rules for graceful degradation of user interfaces. In *Proceedings of the International Working Conference Advanced Visual Interfaces (AVI 2006)*, pages 59–66, New York, NY, USA, 2006. ACM Press. 86

[Gal02] Wilbert O. Galitz. *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 2002. 87

[GW04] Krzysztof Gajos and Daniel S. Weld. SUPPLE: Automatically generating user interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interface (IUI '04)*, pages 93–100, New York, NY, USA, 2004. ACM Press. 88

[Kav07] Sevan Kavaldjian. A model-driven approach to generating user interfaces. In *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 603–606, New York, NY, USA, 2007. ACM. 2

[KBFK08] Sevan Kavaldjian, Cristian Bogdan, Jürgen Falb, and Hermann Kaindl. Transforming discourse models to structural user interface models. In *Models in Software Engineering, LNCS 5002*, volume 5002/2008, pages 77–88. Springer, Berlin / Heidelberg, 2008. 12, 17

[KCV10] Suzanne Kieffer, Adrien Coyette, and Jean Vanderdonckt. User interface design by sketching: a complexity analysis of widget representations. In *EICS*, pages 57–66, 2010. 92

[KFK09] Sevan Kavaldjian, Jürgen Falb, and Hermann Kaindl. Generating content presentation according to purpose. In *Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC2009)*, San Antonio, TX, USA, Oct. 2009. 12, 24

[KKMF09] Sevan Kavaldjian, Hermann Kaindl, Kizito Ssamula Mukasa, and Jürgen Falb. Transformations between specifications of requirements and user interfaces. In *Proceedings of the IUI'09 Workshop on Model Driven Development of Advanced User Interfaces, Sanibel Island, Florida, USA*, 2009. 42

[KRF+09] Sevan Kavaldjian, David Raneburger, Jürgen Falb, Hermann Kaindl, and Dominik Ertl. Semi-automatic user interface generation considering pointing granularity. In *Proceedings of the 2009 IEEE International Conference on Systems, Man and Cybernetics (SMC 2009)*, San Antonio, TX, USA, Oct. 2009. 12, 29

[KRP+10] Sevan Kavaldjian, David Raneburger, Roman Popp, Michael Leitner, Jürgen Falb, and Hermann Kaindl. Automated optimization of uis for screens with limited resolution. In *Proceedings of the 5th International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 2010): Bridging between User Experience and UI Engineering, Atlanta, Georgia, USA*, 2010. 12, 36

[KSS+07] Hermann Kaindl, Michał Śmiałek, Davor Svetinovic, Albert Ambroziewicz, Jacek Bojarski, Wiktor Nowakowski, Tomasz Straszak, Hannes Schwarz, Daniel Bildhauer, John P Brogan, Kizito Ssamula Mukasa, Katharina Wolter, and Thorsten Krebs. Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project, 2007. www.redseeds.eu. VIII, 42, 43, 44, 45, 46

[Lei10] Michael Leitner. Space-saving placement using a structural user interface model. Master's thesis, Technische Universität Wien, Fakultät für Elektrotechnik und Informationstechnik, Institut für Computertechnik, E384, 2010. 74

[LFG90] Paul Luff, David Frohlich, and Nigel Gilbert. *Computers and Conversation*. Academic Press, London, UK, January 1990. 13

[LV04a] Quentin Limbourg and Jean Vanderdonckt. Addressing the mapping problem in user interface design with usixml. In *TAMODIA*, pages 155–163, 2004. 85

[LV04b] Quentin Limbourg and Jean Vanderdonckt. Transformational development of user interfaces with graph transformations. In *CADUI*, pages 105–118, 2004. 85

[MBR07] T. Memmel, C. Bock, and H. Reiterer. Model-driven prototyping for corporate software specification. In *Engineering Interactive Systems: EIS 2007 Joint Working Conferences, EHCI 2007, DSV-IS 2007, HCSE 2007*, pages 158 – 174. Springer, March 2007. 88

[MK08] K.S. Mukasa and H. Kaindl. An integration of requirements and user interface specifications. In *Proceedings of the Sixteenth IEEE International Requirements Engineering Conference (RE'08)*, September 2008. 42

[MLJ06] Francisco Montero and Víctor López-Jaquero. Idealxml: An interaction design tool. In *CADUI*, pages 245–252, 2006. 85

[MM03] Joaquin Miller and Jishnu Mukerji, editors. *MDA Guide Version 1.0.1, omg/03-06-01*. Object Management Group, 2003. 5

[MPS04] G. Mori, F. Paternò, and C. Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Transactions on Software Engineering*, 30(8):507–520, 8 2004. VIII, 81, 82, 83, 84

[MSUW04] Stephen J Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model Driven Architecture*. Addison-Wesley, 2004. 5

[MT88] W. C. Mann and S.A. Thompson. Rhetorical Structure Theory: Toward a functional theory of text organization. *Text*, 8(3):243–281, 1988. 13

[MV08] Benjamin Michotte and Jean Vanderdonckt. GrafiXML, a multi-target user interface builder based on UsiXML. In *Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems*, pages 15–22, Washington, DC, USA, 2008. IEEE Computer Society. 85

[PEPA08] Oscar Pastor, Sergio España, José Ignacio Panach, and Nathalie Aquino. Model-driven development. *Informatik Spektrum*, 31(5):394–407, 2008. VIII, 81, 82, 83, 92

[PEPP08] J. I. Panach, S. Espana, I. Pederiva, and O. Pastor. Capturing interaction requirements in a model transformation technology based on MDA. *Journal of Universal Computer Science*, 14(9):1480–1495, 2008. 88

[PFA+09] Roman Popp, Jürgen Falb, Edin Arnautovic, Hermann Kaindl, Sevan Kavaldjian, Dominik Ertl, Helmut Horacek, and Cristian Bogdan. Automatic generation of the behavior of a user interface from a high-level discourse model. In *Proceedings of the 42nd Annual Hawaii International Conference on System Sciences (HICSS-42)*, Piscataway, NJ, USA, 2009. IEEE Computer Society Press. 1, 13, 41, 84, 90

[PHPC08] Yong S. Park, Sung H. Han, Jaehyun Park, and Youngseok Cho. Touch key design for target selection on a mobile phone. In *Proceedings of the 10th International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI 2008)*, pages 423–426, New York, NY, USA, 2008. ACM. 87

[PKB06] Pekka Parhi, Amy K. Karlson, and Benjamin B. Bederson. Target size study for one-handed thumb use on small touchscreen devices. In *Proceedings of the 8th International Conference on Human-Computer Interaction with Mobile Devices and Services (MobileHCI 2006)*, pages 203–210, New York, NY, USA, 2006. ACM. 87

[PMM05] Angel Puerta, Michael Micheletti, and Alan Mak. The UI Pilot: A model-based tool to guide early interface design. In *Proceedings of the 10th International Conference on Intelligent User Interfaces (IUI'05)*, pages 215–222, New York, NY, USA, 2005. ACM Press. 87

[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. 84

[PVE+07] Inés Pederiva, Jean Vanderdonckt, Sergio España, Ignacio Panach, and Oscar Pastor. The beautification process in model-driven engineering of user interfaces. In *Proceedings of the 11th IFIP TC 13 International Conference on Human-Computer Interaction — INTERACT 2007, Part I, LNCS 4662*, pages 411–425, Rio de Janeiro, Brazil, Sept. 2007. Springer Berlin / Heidelberg. 88

[Ran08] David Raneburger. Automated graphical user interface generation based on an abstract user interface specification. Master's thesis, Technische Universität Wien, Fakultät für Elektrotechnik und Informationstechnik, Institut für Computertechnik, E384, 2008. 2, 12, 41

[Ran10] David Raneburger. Interactive model driven graphical user interface generation. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '10)*, pages 321–324, New York, NY, USA, 2010. ACM. 11, 79, 92

[RBIM97] D. Roberts, D. Berry, S. Isensee, and J. Mullaly. Developing software using OVID. *IEEE Software*, 14(4):51–57, July-Aug. 1997. 86

[RPK+11a] 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)*, New York, NY, USA, to appear June 2011. ACM. 10

[RPK+11b] David Raneburger, Roman Popp, Sevan Kavaldjian, Hermann Kaindl, and Jürgen Falb. Optimized GUI generation for small screens. In Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke, editors, *Model-Driven Development of Advanced User Interfaces*, volume 340 of *Studies in Computational Intelligence*, pages 107–122. Springer Berlin / Heidelberg, 2011. VIII, 37, 39

[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. 80

[Sea69] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, Cambridge, England, 1969. 13

[Sos10] Agris Sostaks. Bringing domain knowledge to pattern matching. In *Databases and Information Systems VI - Selected Papers from the Ninth International Baltic Conference, DB&IS 2010, July 5-7, 2010, Riga, Latvia*, pages 66–79, 2010. 54

[Ste07] Perdita Stevens. Bidirectional model transformations in qvt: Semantic issues and open questions. In *MoDELS*, pages 1–15, 2007. 89

[SV04] Max Schlee and Jean Vanderdonckt. Generative programming of graphical user interfaces. In *AVI*, pages 403–406, 2004. 86

[SV06] T Stahl and M Voelter. *Model-Driven Software Development (Technology, Engeneering, Management)*. John Wiley & Sons, 2006. 5

[Van05] Jean Vanderdonckt. A mda-compliant environment for developing user interfaces of information systems. In *CAiSE*, pages 16–31, 2005. VII, VIII, 5, 8, 81, 86

[Van08] Jean M. Vanderdonckt. Model-driven engineering of user interfaces: Promises, successes, and failures. In *Proceedings of 5th Annual Romanian Conf. on Human-Computer Interaction*, pages 1–10. Matrix ROM, Bucaresti, Sept. 2008. VII, 9

[vH94] M. van Harmelen. Object oriented modelling and specification for user interface design. In *Interactive Systems: Design, Specification and Verification*, 1994. 86

[WSBK08] K. Wolter, M. Smialek, D. Bildhauer, and H. Kaindl. Reusing terminology for requirements specifications from WordNet. In *Proceedings of the Sixteenth IEEE International Requirements Engineering Conference (RE'08)*, September 2008. 43

[XLH+09] Xiangye Xiao, Qiong Luo, Dan Hong, Hongbo Fu, Xing Xie, and Wei-Ying Ma. Browsing on small displays by transforming web pages into hierarchically structured subpages. *ACM Transactions on the Web*, 3(1):1–36, 2009. 87, 88