Master Thesis
Computer Science - University of Twente
Software Technology

Leveraging behavioural domain models in Model-Driven User Interface Development

with

GLUI

Peter Wessels

UNIVERSITY OF TWENTE.

Supervisors
dr. L. Ferreira Pires, University of Twente
prof.dr.ir. A. Rensink, University of Twente
ir. H.V. Nguyen, ING

Juli, 2018

Abstract

Due to the introduction of a wide-range of complex interaction styles and devices, potentially reaching a large and diverse user group, offering a consistent user experience with a user interface has become increasingly complex. Therefore, the traditional approach of implementing user interaction directly into the implementation technology potentially leads to version inconsistency and high maintenance costs.
In this research, we investigate how a Model-Based User Interface Development (MBUID) approach can be applied that leverages the characteristics of behavioural domain models, resulted from a Domain-Driven Design (DDD) approach, to generate verifiable functionality of a front-end application for multiple platforms and different modalities while business analysts with a technical background are able to specify workflows that needs to be integrated.
Complexity is dealt with by separating domain logic from the implementation details. For this process, separate abstraction levels are defined and business logic is encapsulated in behavioural domain models. Three levels of abstraction are defined that increasingly refine the specification of the behaviour of a user interface. We examined how model transformations can be defined in this process to semi-automatically transform the source model to a refined target model while preserving the operational semantics. As the behavioural domain model embodies part of the system behaviour, we examined how we can use these models to define the interaction with the system such that the user can invoke commands to manipulate these models. The combination of the defined models for each abstraction level and the model transformations, the transformation chain, allowed us to examine approaches how behavioural models can be used as a source to generate part of the functionality of an user interface.
To be able to preserve the correctness of each model and to generate a fully operational user interface, we focused on defining a sound specification of the task model, the first abstraction level of MBUID. We leveraged multiple existing techniques to define the structure, data flow and the interaction with the user and the system. This enabled us to create a sound specification and to generate an operational user interface, just as specified in the task model. We used this task model as input for the MBUID process and defined how we can generate applications that contain the same functionality on different platforms. We leveraged characteristics of behavioural domain models twofold. On one hand, we defined patterns to generate separate task models for commands that can be invoked on these domain objects. On the other hand, we defined an approach to combine these separate task models in a composed task model.
The solution has been validated by executing the transformation chain on real-world specifications used in the financial domain to model services.

Contents

Preface

Before you lies the report "Leveraging behavioural domain models in Model-Driven User Interface development with GLUI", the result of a research project conducted at ING. It has been written as part of the final project for the master's programme Computer Science at the University of Twente with *Software Technology* as specialisation. I was engaged in this research from January to June 2018.

This project has been initiated together with Joost Bosman, who intrigued me with his ideas about introducing new software technology in the financial domain. From that point, I was determined to become part of this transition. The innovative spirit and determination of the development team to radically change the software landscape had a significant positive impact on this project.

I would like to thank my supervisors for their guidance and support during the process. From ING, Viet Nguyen: my project benefitted from your enthusiasm about my research and the project. From the University, Luís Ferreira Pires and Arend Rensink: I have always appreciated your constructive feedback on my approach and my report. I want to thank the members of the development team at ING, in particular Kevin van der Vlist, Jorryt-Jan Dijkstra, Jeffrey Bruijntjes and Joery Bruijntjes for their feedback and help during my research.

Finally, I would like to thank my parents for their unconditional support during my study and this final project. I would like to thank my friends and family for their support during the past period. As I conducted my research remotely, I would like to thank my housemates for the daily portion of welcoming distraction. I appreciate it all sincerely.

This project is the endpiece of 7 years of study at the University of Twente. *It has been one hell of a ride, but a good one.* I did not take the easy road, as after my bachelor degree in *Industrial Design*, I followed a completely different master's programme. It has given me a unique background of which I am proud. This master thesis is the result of that.

As always, *stay hungry, stay foolish.*

Peter Wessels

Enschede, June 29, 2018

Chapter 1

Introduction

In this chapter we formulate the objectives and the requirements of the solution, which follows from the problem statement.

1.1 Motivation

The world of software development is changing rapidly. New technologies introduce new possibilities for systems to evolve. Ideally, these technologies should enrich the capabilities of software systems; however, many organisations with large software systems struggle to evolve and maintain their systems. As their systems become increasingly complex, adding new features and using new technologies costs a lot of time and money. Especially in the financial sector, organisations struggle to keep up with new technologies, as new competitors that do not have the burden of having to maintain large legacy systems, are seeking opportunities to offer better services with new technologies.

1.2 Problem Statement

Domain-Driven Design is a method for dealing with complexity of software systems by distilling domain knowledge out of implementation details [1]. By modelling the domain knowledge separate from the implementation, domain experts can use their knowledge and modelling skills to define the characteristics of (complex) domain concepts. This allows complexity to be tackled at the heart of the software (the domain) by the experts who know the domain best.

By capturing both the domain knowledge and the implementation details in models, model transformations can be defined that contains the logic to transform instances of the domain model to instances of the implementation models. By binding the domain knowledge and the implementation details together with model transformations, characteristics of the domain model can be transformed to an implementation of the supporting system. Shifting the focus from writing source code to defining transformations allows new features described in the domain model by domain experts to be integrated with a push of a button.

In this research, we focus on the development of a multi-user front-end application for a reactive system by using domain-driven design in combination with a model-driven approach. We define a front-end application as an operational user interface that handles both the communication with the back-end as well as the interaction with the end-user. A multi-user front-end application is defined as an environment in which different types of users interact with each other with a certain goal, for example, a business goal. The problem is that user interface development has become increasingly complex and the traditional approach of developing a user interface can potentially lead to version inconsistency and fails to deliver a consistent user experience among different devices and platforms [2].

Various research efforts have been devoted to MBUID. This approach to User Interface (UI) development supports the integration of characteristics of domain concepts in a user interface and deals with the complexity of UIs development. These methods, however, have been developed with *static* domain models, while a domain-driven approach does necessarily define how domain models are modelled, that is specific to the domain. Thus, domain models defined in a domain-driven approach are not necessarily aligned with MBUID, leaving room for improvement.

Current MBUID approaches mainly use a *descriptive* or a *relational* domain model that defines *static* characteristics and relations between concepts. A *behavioural* domain model also includes possible interactions with domain concepts defined as, for example, a state machine that defines different states and transitions of a domain concept. For the development of a front-end application, this offers opportunities regarding the verification of specified behaviour as well as leveraging a behavioural model as a source for (semi-)automatically obtaining the required UI functionality. In this research, we define methods to exploit the characteristics of a behavioural domain models to generate verifiable functionality of a front-end application.

1.3 Objective

Our main objective has been to investigate how to apply a MBUID approach that uses characteristics of behavioural domain models such that the application correctly implements the specified behaviour. To achieve this, we investigated the features of the involved models and model transformations with the following requirements:

Requirement 1. The generated user interface should be operational such that it facilitates the communication with both the user as well as the system.

Requirement 2. The input modelling language allows business analysts with a technical background to produce a software specification of a user interface for different kinds of users.

Requirement 3. Each intermediate model of the transformation chain refers to behaviour of a domain model such that the resulting user interface integrates this behaviour.

Requirement 4. Instances of intermediate models can be verified at design time with respect to both the defined domain concepts as well as user interface logic. The user interface does not allow illegal actions neither should it obstruct users to perform valid actions, as defined in the domain model. The correctness the input model should be verified with respect to the domain model; does the task model complies with the input constrains as defined in the domain model?

Requirement 5. The structure of the transformation chain should account for changes in interaction styles, devices and platform. Developing a new application should not require rewriting existing software components, nor the software specification, in case the product characteristics, the domain knowledge, remain the same. Instead, new model transformations and dedicated intermediate models should be introduced to support the new platform or device.

1.4 Validation

To validate the requirements of the solution we defined for each requirement a principle that determines if the solution is conform our requirements.

Requirement 1. The generated user interface should define a clear interface with the system as well as presenting the user with the appropriate tools to finish the described tasks.

(a) The implementation of the user interface should contain a description of the interface that consists of the input and output conditions of each interaction with the system.

(b) The user interface should contain the user interface construct elements as such that the user control the behaviour of the system as well as input and manipulate data.

Requirement 2. Business analysts should be able to define and validate the specification for the user interface autonomously.

Requirement 3. Each intermediate model should define a description of *which* characteristics of the domain model and *how* it is used to determine behaviour of the user interface.

Requirement 4. The input as well as the intermediate models should be constraint as such that only *valid* models are accepted. Non-valid models should be rejected as they do not result in a correct implementation. Validation methods should be able to asses the validity of these models.

Requirement 5. Introducing a new platform, device or interaction style should solely involve the definition of new model transformations and possibly new (dedicated) intermediate models.

1.5 Structure & Approach

The structure of this report reflects the approach that has been followed during the research. Figure 1.1 depicts the structure of transformation chain labelled with the chapters in which we discuss the features and involved design choices.

In our approach, we first conducted a literature review in which we analysed the concepts involved in this research. Then, we defined the assumptions on the basic structure of solution, the specification and behaviour of the considered system, and our approach to ensure correctness of the resulting solution. Hereafter, we defined the metamodel of the input modelling language and the intermediate user interface models and the model transformations involved in the the process of generating an implementation. Then, we defined approaches to use behavioural domain models as input for the task model. Parallel with these steps, we defined approaches to verify the correctness of the involved models. Finally, we have validated our approach to verify to what extent the solution solves the problem at stake.

Chapter 2 introduces the main concepts of this research to facilitate understanding by the reader. Additionally, the most prominent MBUID approaches has been reviewed.

Chapter 3 presents features of the **transformation chain**. We define the concepts of each model, an approach to achieve correctness, and how models in the transformation chain can refer to properties of the system on a conceptual level.

Chapter 4 focuses on a input specification language for the **task model**. We introduce a notation to describe workflows, data flow, as well as a method to define the interaction with the system and the user.

Chapter 5 presents **user interface models** that refine the task model. We define the involved concepts and the involved model transformations.

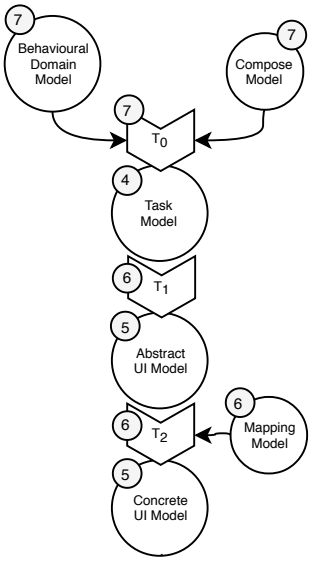


Figure 1.1: Overview of the transformation chain labelled with the chapters where the models and model transformation are discussed

Chapter 6 discusses the **model transformations** between models of the transformation chain.

Chapter 7 describes how we can use **behavioural models** to generate parts of the task model.

Chapter 8 discusses methods to verify the **correctness** of the resulting user interface.

Chapter 9 discusses the experiments that have been carried out to **validate** the transformation chain.

Chapter 10 concludes with final remarks and future work.

Chapter 2

Background

In this section we provide background information to familiarise the reader with the subject. We highlight the different aspects of the problem at stake.

2.1 User Interface

The User Interface (from hereon referred to as UI) in Human-Computer interaction is the mechanism wherein the user can interact with the machine. These interactions allow the user to operate and control the machine while the machine gathers valuable feedback that can help the user. As this definition is very broad, many types of user interfaces exist. We discuss the major types of user interfaces:

- **Graphical User Interfaces (GUI)** presents a graphical representation of the information and the possible controls.
- **Command line interfaces (CLI)** enables users to interact with the system by giving the system commands in a textual form.
- **Virtual Reality User Interfaces** enables users to interact with the system in a virtual world in a 3D environment.
- **Tangible User Interfaces** build upon human skills to manipulate and sense the physical world by integrating the digital and physical world [3].
- **Voice Assistent User Interfaces** enables users to interact with the system via command-like instructions communicated via voice. WA Voice Assistent User Interface is often combined with a GUI, to give feedback in a graphical manner, whereas devices without such feedback mechanisms give feedback via voice.

As the UI is a gateway in which a user can interact with the application, designing a UI requires the designer to cope with the complexity of both the application and the user [4].

Questions to be answered are: *Which operations are available to the user? How does the user perceive the information presented to him? How does the user react on feedback?*

Considering that a designer must deal with both worlds (the user and the system) it is not surprising that research shows that UI development of an interactive system has become more time-consuming and therefore more costly. On average the development of such a UI represents 47% of the source code, requires about 45% of the development time and 50% of the implementation time, and covers 37% of the maintenance time [5].

To tackle the problem of designing a UI, different approaches exist. Among others, Vanderdonckt has distinguished 4 major approaches in User Interface development [6]:

1. **Traditional approach** In this approach, the developer develops a UI by composing views, e.g., windows and web pages, with user interface components, e.g., buttons, forms and titles. When the functions of the system are developed, these views are expanded such that the UI can call system functions and become operational.

2. **Programming by demonstration** By adding actions to the UI the developer can demonstrate how the UI interacts with the user. This approach is very similar to the traditional approach, except that it adds the possibility to assess the usability of the UI at design-time.

3. **Model-based approach (MBUIDE)** By separating UI-related concerns into formal declarative models a functioning UI can be generated. Once each model is defined, the code generation process can be automated.

4. **Task-based approach** Very similar to the previous approach is the task-based approach except that the task-model is first specified. Using a task-model, other models can be derived, refined or specified.

The development of user interfaces has become more complex due to some serious challenges [2]. Vanderdonckt presents an analysis of variables that are at the root of this increase of complexity.

- **Diversity of users.** An interactive system can no longer consider users to be similar, as they show differences in terms of skills and expertise with regard to operating an interactive system through a user interface.

- **Richness of cultures.** When dealing with applications that are globally accessible, the UI cannot remain the same for each culture, as cultures can have different languages, different customs or even different demands of the UI.

- **Complexity of interaction devices and styles.** Due to the availability of a wide variety of interaction devices and styles, the handling of events generated by these devices requires programming skills that can go beyond the capabilities of an average developer of an information system.

- **Heterogeneousness of computing platforms.** In the world of software technologies, new platforms and technologies will be introduced continuously, posing new limitations and constraints on the UI.
- **Multiple working environments.** Users should be able to interact with the user interface under different circumstances (e.g. light and sound conditions).
- **Multiple contexts of use.** The context in which a user operates can change. For example, if the user decides to start working on his computer and continue working on a task on a mobile device, it would be convenient that the application can adapt to this context change.

Even though some interface development techniques such as universal design [7] and inclusive design [8], promote designs that fit for the largest possible population, the UI cannot longer be considered as independent of its usage context [9]. This usage context can be defined as a triplet of a user, platform and environment and determines the characteristics of the UI in different contexts [10]. With these challenges in mind, the traditional approach for the design of user interfaces would require many versions of the UI. This potentially leads to version inconsistency and therefore high maintenance costs [11]. Adaptive UIs have been promoted as a solution to offer a consistent UI in changing contexts, as they automatically adapt to the context of use at runtime [9].

2.2 Model-Driven Engineering

Model-Driven Architecture (MDA) is a software development approach in which models have a central role [12]. By structuring specifications expressed as models, transformations can be defined to automate the implementation of the system. The approach is based on separating application domain knowledge and application logic from the underlying platform technology. The basic pattern is to define a Platform-Independent Model (PIM) that captures domain knowledge and a transformation to a Platform-Specific Model (PSM) that maps the domain knowledge to platform-specific implementation details. MDA claims to properly deal with the complexity of large systems and the interaction and collaboration between organisations, people, hardware and software.

2.2.1 Model-Driven User Interface Development

Since the 1980s, in the field of UI development, research has been carried out to use the Model-Driven approach to structure the development of a UI. In this field, four generations of Model-Driven User Interface Development (MDUID) approaches can be distinguished [13].

The first generation was motivated by the idea of using one universal UI model that integrates the relevant aspects of a UI. The second generation of systems can be characterised by abstracting aspects of the UI model in separate high-level models like, e.g., dialog, task and presentation models. The introduction of new mobile devices like smartphones and PDAs motivated the third generation of MDUID approaches. The current fourth generation of MDUID approaches focuses on the development of context-aware user interfaces that have the ability to adapt to the user, platform and environment.

The Cameleon Reference Framework (CRF) is a fourth generation MDUID approach and has become widely accepted in the Human Computer Interaction Engineering community as an approach to structure the development of UIs supporting multiple contexts of use. The framework adopts a model-based approach by prescribing the development of UIs based on three *Ontological models*. These ontological models express context-of-use configurations, domain concepts and adaptation dimensions.

1. The *Context of use model* describes the characteristics of the UI for different users, platforms and in different environments. For each context of use dimension, a corresponding model can be defined.

2. The *Domain model* expresses the domain objects that can be manipulated by the user in tasks. These tasks refer to activities with a certain goal that can be performed by the user with the system.

3. The *Adaptation model* expresses how the UI should react if the context of use changes. It also contains an *Evolution model* that denotes how the UI should evolve into a new UI.

These three models are the foundation of the CRF framework and can be used at the different abstraction levels that the framework defines.

Abstraction level 1. Task Model The task model represents the highest level of abstraction of the UI. This model expresses the task descriptions produced by the designers for that particular system and context of use. In these models, the UI is abstracted from the implementation details and modality (voice, graphical, gestural, etc.) and presents the hierarchy of tasks the user has to perform to reach a certain goal.

Abstraction level 2. Abstract User Interface (AUI) The AUI expresses the rendering of the tasks and domain concepts defined in the previous level independent of any modality and implementation details. The Abstract UI consists of a collection of AUI Units with relationships among each other to specify the navigation.

Abstraction level 3. Concrete User Interface (CUI) The CUI refines the AUI by adding information on how the UI has to be perceived and manipulated by the user. This model adds the notion of modality and is, therefore, modality dependent. The UI is specified in terms of the layout, positioning of the widgets and the interface navigation. The look and feel of the UI is also specified in the CUI.

Abstraction level 4. Implemented UI The implemented UI uses presentation technology such as HTML, Swing and Motif to describe the UI at a specific platform and device. The UI can either be compiled or interpreted such that different targets can render the UI.

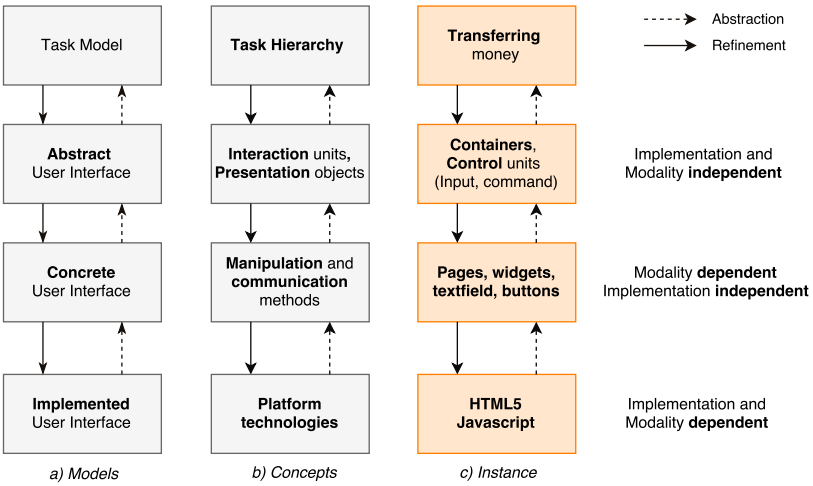


Figure 2.1: Schematic overview of the CRF structure

2.2.2 *Domain-Driven Design*

Domain-Driven Design (DDD) is a software engineering approach to connect an implementation to complex domain logic in evolving models [1]. This allows both technical and domain experts to analyse iterations of the domain model. Multiple approaches to DDD exist, such as the Functional Approach [14] and the Object-Oriented approach [1]. Domain-Driven Design is often implemented using a Model-Driven approach, as the domain knowledge can be captured in a model, and be transformed to an implementation.

2.3 User Interface Description Languages

To express the concept at the different abstraction levels of the CRF, languages have been defined to express the concepts modelled in the first 3 abstraction levels. An overview of User Interface Description Languages (UIDLs) compliant with the CRF is shown in Figure 2.2.

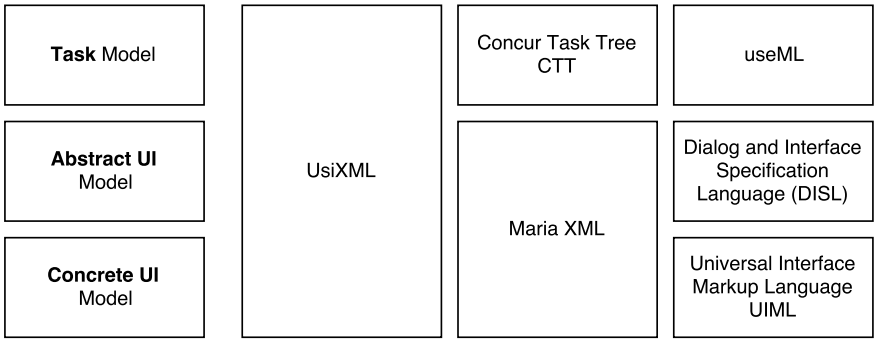


Figure 2.2: Compatible UIDLs with the CRF abstraction levels.

- The USer Interface eXtensible Markup Language (UsiXML) [15] is a XML-based language that is capable of expressing concepts of the first 3 CRF abstraction levels. UsiXML uses graph theory to formalise the language. As a result, an instance of the UsiXML metamodel is a directed, typed graph.

- To capture the semantics of a task, the Concur Task Trees (CTT) notation can be used to define a hierarchical task structure with a wide range of (temporal) relationships to constrain the execution order of (sub)tasks [16].

- Maria XML is a general purpose language and able to express both the Abstract User Interface and the Concrete User Interface model [17]. Maria XML differs from UsiXML as is not based on a graph structure, and the metamodel of the concrete user interface is different from the UsiXML metamodel as the authors made different design choices.

- Useware Markup Language (useML) [18] was originally developed to support a user-centric development process by providing a language that allows task modelling and analysis. A *use model* (task model) consists of platform-independent tasks modelled as so-called use objects in a hierarchical structure. This model is structured as a tree. The leaves represent elementary Use Objects (eUO). A eUO is an atomic interactive task. Available types eUOs are inform, trigger, select, enter and change. Like UsiXML and Maria XML, useML (version 2.0) supports temporal relations to relate eUOs.

- The User Interface Markup Language (UIML) is a XML-based, declarative language designed for specifying a canonical XML representation of any UI. UIML follows the structure of the Meta-Interface Model of [19], which divides the interface into 3 separate components: presentation, logic and interface.

- Dialog and Interface Specification Language (DISL) [20] is an extended UIML subset designed to model the abstract user interfaces, focusing on supporting adaptation, scalability, reusability, usability for developers (tools) and low resource demands.

Two approaches can be used to define the *final user interface*: the UI can either be interpreted using a User Interface Management System (UIMS) or compiled using some platform-specific presentation technology. An interpretational approach, like MASP [21], DynaMo-AID [22] and Supple [23], uses models to render a UI at runtime. Model interpretation at runtime is usually more suitable for supporting adaptive behaviour than relying on static code artifacts [9]. Another advantage is that UI adaptations can be deployed without recompiling the application. A compiled approach relies on code artifacts in a specific presentation language. These artifacts are generated at design-time from the CUI, and adaptive behaviour is limited at runtime. An advantage of this approach is that many platforms can be supported if they support a certain way of defining interactive and presentation behaviour in code. A combination of these two approaches is using models defined at design-time to generate code artefacts at runtime. The modelling approach used by the 3-Layer Architecture [24] is an example of this approach. Whereas an interpretational approach seems to have more advantages than a compiled approach, research showed that performance can be an issue in worst-case scenarios [25]. Also, existing tools support is limited and not integrated in a mature IDE.

2.4 Model Transformation Languages and Technologies

In Model-Driven Engineering (MDE), model transformations are used to transform concepts from the source model to the target model. A commonly used approach is to define a model transformation based on the source and target metamodel. The structure of this technique is shown in Figure 2.3.

According to Object Management Group (OMG) standards, a metamodel is a *special* kind of model that specifies the abstract syntax of a modelling language [27]. A model transformation definition defines the procedure to transform concepts defined in the source metamodel to concepts defined in the target metamodel.

We distinguish horizontal and vertical transformation. Whereas in a top-down approach vertical transformations refine the model by increasing details about the target model, we define transformations at the same level of abstraction as horizontal transformations. Horizontal transformations are used to refactor, complete, or optimise a model to improve the internal structure and/or quality [28].

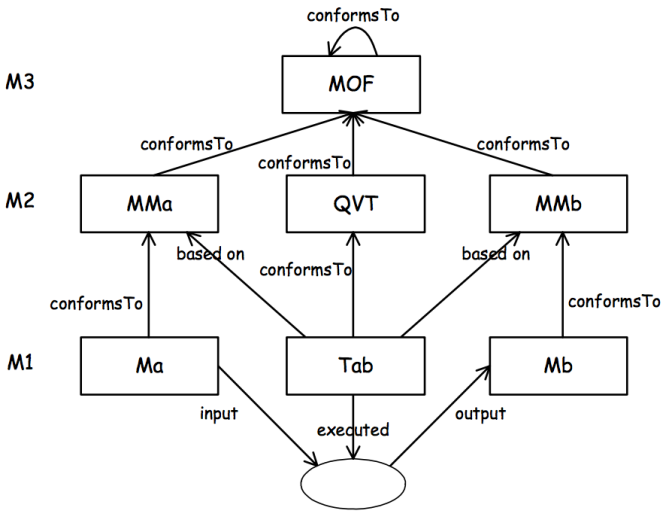


Figure 2.3: A model transformation approach based on the metamodels of both the target and source model as published in [26]

In a top-down approach in Model-Driven Architecture, vertical transformations bridge the implementation gap from specification to an executable system. In a bottom-up approach, a transformation omits details to create abstractions from detailed models. A bottom-up approach can be used in *Reverse Engineering* to inject a model from an existing implementation [29]. Transformations do not necessarily have to be defined to support a single direction, from source to the target model. By defining transformations in both directions, synchronisation between emerging models at different levels of abstraction can be achieved [30].

At the top level, model transformation approaches can be classified in two major categories: model-to-text and model-to-model transformations. We consider programming code as well as other forms of text, such as SQL queries and system configuration files, as model-to-text approaches. In model-to-text transformations, we consider 2 approaches: Visitor-Based and Template-Based approaches:

- **Visitor-Based Approaches** A basic approach to model-to-text transformations is a visitor-based approach that generates code while traversing the inner structure of the source model. An example of this approach is Jamba [31].
- **Template-Based Approaches** A more popular approach is the template-based approach that uses parametrised templates of the target system. A template consists of code snippets where information from the source model can be injected. Examples of techniques that use this approach are JET [32] and AndroMDA [33].

When the semantic gap between a the source and target models is large, multiple model-to-model transformations can be defined to bridge this gap. The advantage of using model-to-model transformations is that these transformations tend to be more maintainable, modular, and can be used to debug the transformation then a single transformation. In addition, model-to-model transformations can be used to define horizontal transformations and thereby supports synchronisation between models. We consider 4 approaches to model-to-model transformations:

- **Direct-manipulation approaches** These approaches use an API to access the internal structure of the model. An object-oriented framework, like SiTra for Java [34], is an example of this approach. However, features like scheduling and generating traces have to be implemented from scratch, making it unsuitable for large complex transformation [35].

- **Relational approaches** A relational approach is declarative, where the main concept is a mathematical relation. The basic idea is to relate a source to a target element type and define constraints of this relationship. Logic programming can be used to implement the relational approach. An example of a relational approach is [36]. Akehurst and Kent discusses an approach that captures the essence of mathematical relations in a metamodel.

- **Graph-Based approaches** Graph transformations can be used in a model-driven approach. These approaches typically operate on typed, attributed, labelled graphs. The main concept of graph transformation is to define a rule consisting of a pattern in the source model and a pattern in the target model. When a pattern in the source model is matched, the model is transformed and replaced by the defined pattern in the target model. Conceptually this approach is the same as a metamodel-based transformation as both approaches map concepts from a source to a target model. The metamodel approach, however, does not constrain the structure of the models.

- **Structure-Driven Approaches** A structure-driven approach distinguish 2 phases: in the first phase the hierarchical structure of the target model is created, in the second phase the attributes and references are added.

Hybrid approaches combine these approaches, allowing developers to choose the most appropriate characteristics of each approach depending of the task.

2.5 Formal Software specification

Since the beginning of Computer Science, formal specifications have played an important role. Whereas behaviour of a system can be specified in natural language, Meyer was among the first to demonstrate the deficiencies of a requirements document written in natural language [37]. Such an informal approach suffers from problems regarding noise, silence, overspecification, ambiguity, contradictory statements, forward referencing and wishful thinking. These problems are hard to solve in natural language alone. Therefore, various research efforts have been devoted to formally specifying software systems.

A Formal Specification Languages (FSL) uses mathematical concepts and notations to express the behaviour of a system. Sets, functions and variables can be used to express properties that a system should satisfy. FSLs have been developed to describe what a system must do without specifying how it should be done. Since *ambiguity* is a key source of errors, as it allows members of the development and validation team to interpret requirements differently, formal specifications are useful as they specify behaviour in an unambiguous manner [38]. Different formal languages exist and each of them uses their own approach. King distinguished formal language techniques in two groups: the model-oriented and property-oriented techniques [39].

Property-oriented techniques describe a system indirectly by stating properties about it. The declaration of such properties constrains the number of models that satisfy these properties, i.e, the correct programs. Different approaches of property-oriented approaches are Algebraic Methods, Model Logics and Axiomatic Methods.

Model-oriented Instead of constraining the number of models, model-oriented techniques define a model to represent a correct program. In this case, a program is correct if it behaves the same as the specified model [40]. Among different approaches *transition-oriented* and *state-oriented techniques* use the model-oriented approach.

An example of the use of formal software specification is to define the behaviour of a financial service within a software system. By using a model-oriented approach, such an entity can be modelled as a finite state machine that describes the behaviour and the possible transitions in different states [41]. Business processes can be specified as a sequence of interaction and manipulation of these services. Such a specification can serve multiple purposes, e.g, as a communication instrument, to domain experts, to the developers and to the testers or as input of a toolchain that transforms the specification to an implementation of the system. By using a specification in a graphical or textual format, the characteristics of an individual service can be analysed and verified before building the software component that implements the supporting functionality, which is a costly process.

2.6 Software Verification

Developing a software system is often a process of requirements engineering, interpreting requirements and mapping requirements to functionality, implementing the functionality and testing the result. Whereas software often doesn't behave as expected from the requirements, verifying software is often a necessary step. The goal of software verification is to verify if software behaves conform the requirements. We define two major classes in software verification: *software testing* and *formal verification*. The former involves the execution of tests to detect defects until one has enough confidence that no defects exist. The latter involves theorem-proving, the process of showing that the program matches the specified function.

2.7 Conclusion

As we have discussed several approaches of User Interface development, the traditional approach shows serious issues regarding consistency and maintenance in the development of user interfaces for different modalities and platforms. An approach to tackle this complexity is Model-Driven User Interface Development, which is based on Model-Driven Engineering. This approach tackles the complexity of the development of user interfaces by separating the domain knowledge from the implementation details. The Cameleon Reference Framework implements this MBUID approach and provides a basic approach to structure concepts at different abstraction levels to generate a user interface for different modalities with consistent functionality across platforms. Different modelling languages have been developed that implement the CRF approach. As we require the correctness of the user interface to be verified, we use formal specification techniques.

In this research we define an approach that uses the CRF framework as the basic premise. We define how the intermediate models can be structured such that we can generate an operational user interface (Requirement 1), the highest level of abstraction can be specified by business analysts (Requirement 2), behaviour of behavioural domain models can be integrated (Requirement 3) and correctness can be verified (Requirement 4) while retaining the feature of the CRF framework to support the development for a functional consistent user interface for different modalities and platforms (Requirement 5).

Chapter 3

Transformation chain

In this chapter we define the general structure of our transformation chain. We give an overview of the involved concepts at each level of abstraction and the main purpose of each transformation. We define the conceptual model of the system and our approach how to achieve correctness.

The transformation chain forms the spine in our approach. As we follow a model-based user interface development approach, the transformation chain consists of models connected with model transformations. These models contain an explicit and mostly declarative description of the presentation and the behaviour of a user interface for an interactive system. Model transformations define the rules to transform concepts from the source model to concepts of the target model. This allows models to be abstracted, when information is omitted with a specific goal, and models to be refined, when information is added.

3.1 Overview

We adopt a common approach of the model-driven user interface development paradigm by structuring the transformation chain in 4 distinctive abstraction levels: the task model, abstract user interface, concrete user interface and the final user interface, as discussed in Section 2.2. Each level of abstraction serves its own purpose in this top-down approach. We discuss the relation between these abstraction levels.

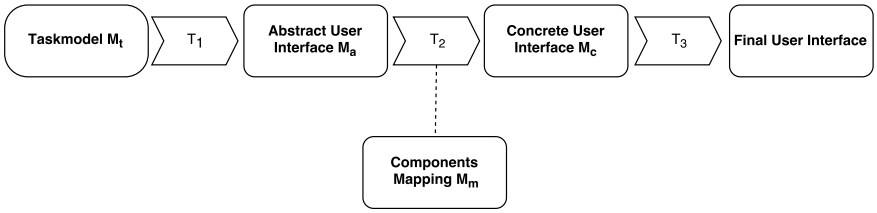


Figure 3.1: Models and transformations in the transformation chain

The highest level of abstraction of a user interface, the *task model*, contains a description of the required user interaction from a user perspective. Tasks can be sequenced in a specific order to define a *workflow*, the order in which tasks have to be performed to achieve a certain (business) goal. The *abstract*

user interface refines these tasks into user interface components and behaviour, abstracted from modality and implementation technology. Only high-level interactions, e.g., presenting information and requiring input from the user are supported with high-level construct elements because of this abstraction. Behaviour is specified as a reaction to specific interaction on these elements and can consist of validation rules to check if the interaction is conform the specification. For example, if the user presses a button, the specified behaviour can react by executing the next task, as specified in the task model, if and only if the input given by the user is validated. The *concrete user interface* refines the abstract user interface and specifies the presentation and behaviour for a certain modality, e.g., a graphical user interface. It refines the low-level construct elements to concrete construct elements, like e.g., textfields and buttons for a graphical user interface. Because a concrete user interface model is specific for a modality, dedicated models are defined for each modality. For example, a graphical user interface consists of graphical elements like a menu, forms, textfield and buttons whereas a voice user interface can consist of a dialog. The *final user interface* is implemented in the chosen implementation technology. Depending on this technology the semantics are captured in a model which can be translated to executable code or interpreted on runtime.

3.2 Conceptual model of the system

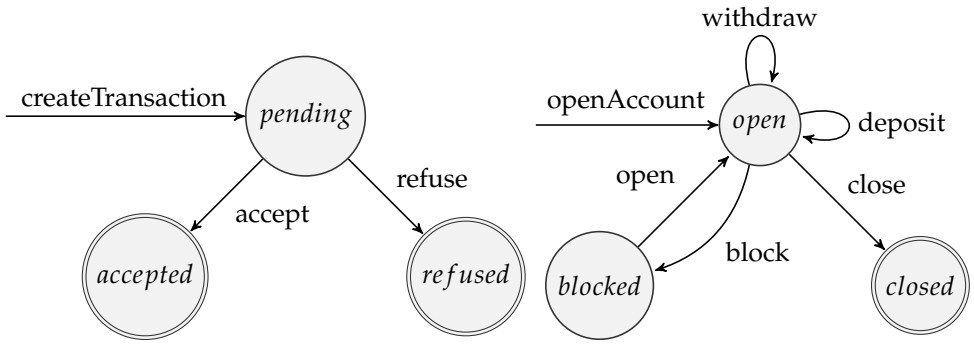
The result of the transformation chain consist of an operational user interface, an application that enables the user to interact with the system. To define what characteristics of the system we can refer to in the four models, we analyse and define the characteristics of a system that consists of behavioural domain models. From this model we can derive the functionality that is at the disposal of the user and can be referred to in the transformation chain.

3.2.1 Concepts

In this research, we define a conceptual model of a multi-user reactive system as a set of *Objects* that can be manipulated by *Actors*. We define two types of actors: internal and external actors. Internal actors are objects within the system that are capable of manipulating other objects. External actors can be a (specific type of) user or other systems. Objects have attributes that contain information about their state. These attributes can also contain links to other objects. In our conceptual model, objects are stateful, which means that the state of the object determines the valid *Operations*. Operations are means to manipulate the state of the object as well as manipulating the state of linked objects. Actors in the system can have a set of capabilities. A *Capability* defines which operations the external actor can trigger.

3.2.2 *Object Behaviour*

We can define the behaviour of an object, that is, the valid operations in a specific state, as a state machine. We can define a state machine as a directed graph where nodes denote states and edges denote transitions. A transition defines an atomic action that changes the state. A transition can require arguments that can be constrained by preconditions. Postconditions define how the attributes of the object are changed and which external objects are manipulated. So, apart from internal state changes, objects can trigger transitions of other objects. The start transition determines the initial state of the state machine. Apart from the initial state, we distinguish end states, which are states with no outgoing transitions. We call the process of objects from initial states reaching end states a life cycle. Since objects can have multiple end states, an object can have different life cycles. Figure 3.2 depicts the state machines of two fictive behavioural domain models.



(a) Transaction (b) BankAccount

Figure 3.2: State machine of the domain model *SimpleAccount* and *Transaction*

3.2.3 *Actors*

In a system we define users with different sets of capabilities. The set of capabilities is the subset of the set of operations in the system. In this set of capabilities, we allow two types of operations: commands and queries. The former allow the user to manipulate the state of the system whereas the latter retrieve information about the state of the system. Since the system is stateful, it depends on the state of the system which operations are valid.

3.2.4 *Example: Transaction processing*

To illustrate the conceptual model, we look at an example of a system that can transfer money from one account to another, shown in Figure 3.3. We use the defined behaviour of the *BankAccount* object and the *Transaction* object from Figure 3.2 specified in Listing 3 and 4 respectively, and the specification of the system in Listing 1. From the specification of the objects, we generated the set of operations and defined the relevant queries as operations, suffixed with .View(). We have assigned a set of capabilities per user role.

In the example, the system consists of three objects: a *Transaction* object, a *Bank account A* object and a *Bank account B* object. The set of capabilities and the state of the object define the valid operations of a user on these objects. To create an instance of a transaction object, the user has to have the operation *createTransaction* in his set of capabilities. This initial condition and the fact that the object has to be initialised, determines if the user is able to trigger the transition. To actually invoke the operation, the provided arguments of the transition, *from*, *to* and *amount*, has to meet the preconditions the transition *createTransaction*.

Whereas objects can invoke transitions of linked objects, the preconditions of the linked transitions has to be met. In this example, the *Accept* transition triggers the *Withdraw* and *Deposit* transitions. Both transitions require the state of the object to be *open* and the former requires that the balance should be sufficient.

If all these checks are successful, the transition can be triggered and the state of the *Transaction* object changes to *pending*. In this state, the life cycle of a transaction is not completed as the transaction has to be accepted first. Another user, in this example User X, can call *AcceptTransaction*, as its set of capabilities allows it to do so and the *Transaction* object is in the state the transition can be triggered. This changes the state of the object to its final state, which is *accepted*.

The state of the user interface depends on the capabilities of the user and the state of the relevant objects. To account for every state we could define an instance of a user interface model for each state of the relevant objects. In theory, this would allow us to validate if in every state, the user interface would be correct. However, when the system consists of more then a single object and user, than the states of the user interface can grow significantly. A more logical approach would be to define a model for each type of user, typical for each role, in a role based environment. A role determines the set of capabilities of a specific type of user. This approach allows us to assess the completeness of the definition of the user interface, in the sense that the user interface covers all relevant operations to complete a process. For example, each object defines a certain life cycle that denotes when all business processes of the object can be considered as completed. In the context

$$
\begin{aligned}
Objects &= \{Transaction, BankAccountA, BankAccountB\} \\
Actors &= \{UserA, UserX\}
\end{aligned}
$$

$$
\begin{aligned}
Operations &= \{Queries \cup Commands\} \\
Commands &= \{Transaction.CreateTransaction(from, to, amount), \\
&\quad Transaction.AcceptTransaction() \\
&\quad BankAccount.Withdraw(amount) \\
&\quad BankAccount.Deposit(amount)\} \\
Queries &= \{BankAccount.View() \\
&\quad Transaction.View()\}
\end{aligned}
$$

$$
\begin{aligned}
Capabilities(UserA) &= \{Transaction.CreateTransaction(from : BankAccountA, to, amount) \\
&\quad BankAccountA.View()\} \\
Capabilities(UserX) &= \{Transaction.AcceptTransaction()\}
\end{aligned}
$$

Listing 1: Definition of the *Transactions processing* system

$$
Capabilities(UserA) \cup Capabilities(UserX) = Operations \qquad false
$$

Listing 2: Formula to verify if the total set of capabilities comprises the set of operations

of a transaction, the object can be considered as complete when the transaction is accepted. A bank account is completed when the bank account is closed. In these states no outgoing transitions can be triggered. If we define the capabilities of each user, we can verify if the total set of capabilities covers a complete life cycle: does the set of capabilities consist of all operations needed to execute a path from initiating an object towards all end states? This property does not necessary assesses if the user interface is correct. Whereas multiple applications can invoke transitions, and not necessarily only the user interface, this property does not always have to hold. It enables the modeller to assess if he included the necessary operations. We can check this property for each user, and the life cycle of an object.

Throughout the transformation chain we refer to characteristics of the domain models. The set of capabilities is used to generate a task model for each role. The defined operations define which arguments need to be provided by the user. The queries define which information the system can provide the user interface. Either to use this information to determine UI behaviour, e.g., to determine if the object is in a valid state such that the user can start a particular task, or to present this information to the user.

```
event withdraw[] (accountNumber: IBAN, amount : Money) {
     preconditions {
         amount >= balance;
     }
     postconditions {
         new this.balance == this.balance' - amount;
         new this.accountNumber == accountNumber;
     }
}

event deposit[] (accountNumber: IBAN, amount : Money) {
     preconditions {
     }
     postconditions {
         new this.balance == this.balance' + amount;
         new this.accountNumber == accountNumber;
     }
}
```

Listing 3: Specification of withdraw and deposit events as defined for the BankAccount object

```
event CreateTransaction[] (from: IBAN, to: IBAN, amount : Money) {
     preconditions {
         amount >= EUR 0.00;
     }
     postconditions {

     }
}

event AcceptTransaction[] () {
     preconditions {
     }
     postconditions {
    from.withdraw(amount);
    to.deposit(amount);
     }
}
```

Listing 4: Specification createTransaction event as defined for the Transaction object

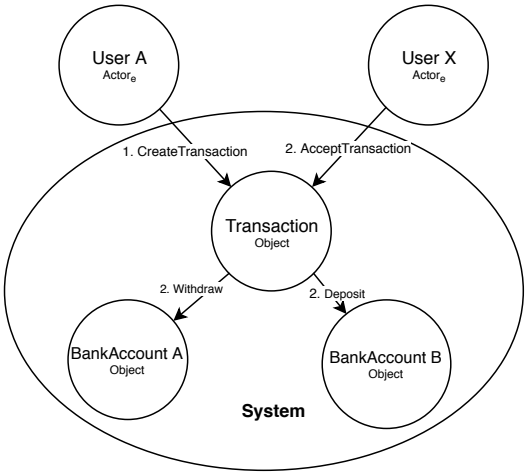


Figure 3.3: Instance of the conceptual model

3.3 Correctness by Construction

We require the output of the transformation chain, the implementation, to be functionally correct with respect to the input of the transformation chain, the task model. The implementation is correct if it behaves as expected. Since the implementation is often a human task, the introduction of errors is often unavoidable. Therefore, testing is a necessary task to validate if the implementation conform to the requirements and if no errors have been introduced. Two issues can arise here. If developers, business analysts, testers and other stakeholders interpreted the requirements differently, qualifying the software as correct is not possible. The other issue is that testing is often an engineering task. The more effort is put into this task, the more errors can be revealed. However, as a result, there can be no guarantee that no errors exist in the system-under-test as more effort could lead to revealing more errors. Our approach to these issues is essentially different than common development practises. We seek to create an implementation that is initially correct, by applying the method of Correctness by Construction (CbyC) as described by Hall and Chapman [38]. CbyC defines two fundamental principles to software engineering - *to make it difficult to introduce defects in the first place, and to detect and remove any defects that do occur as early as possible after introduction* [42]. By integrating the building blocks of CbyC into the structure of the transformation chain, the generated implementation is, in theory, correct.

3.3.1 Building blocks of Correctness by Construction

This may sound ambitious or unrealistic to be used in practise. However, with the right building blocks we can give certain guarantees to the implementation. In the transformation chain we adopted the following building blocks.

Unambiguous notations

The cornerstone of this method is the use of an rigorous notation for all deliverables. A rigorous notation is capable of capturing the relevant characteristics as well as defining these characteristics as unambiguous as possible. This allowed us to assess the properties of each intermediate model in the transformation chain. In this research we focus on a rigorous notation of the task model while preserving the properties of this task model in the transformations.

Strong Validation

Ideally, for every human involvement in the transformation chain, we want to validate the properties of the defined artefact at design-time to avoid the introduction of errors. This principle is essential in our approach. We want to detect errors when they are introduced. Since formal methods can provide rigorous notations, we can use formal methods to validate the correctness of the defined artefact. In our approach the task model is based on an Operation Petri-net (OPN). This allows us to validate the task model by simulation and by checking correctness properties, such as: does it express all the workflows that the modeller envisioned, and is the specification consistent?

Correctness preserving transformations

Since we defined the use of sound notations for the task model, and the existence of validation tools to validate the correctness of this model, we defined transformations that preserve correctness. We shift the focus from creating an implementation by hand to generating an implementation for every instance of the task model, we achieve correct implementations by validating the transformations. We define properties that should be preserved. If these properties are preserved during the entirety of the transformation chain, including the generated implementation, we conclude that the transformation chain produces a correct user interface with respect to the defined properties.

Avoidance of Repetition

The structure of the transformation chain facilitates the reuse of the models at different abstraction levels. Therefore, we can use existing models to create a user interface for a new platform. Also, since domain models are defined in separate models, and we define references to these models, we avoid redefining domain model behaviour.

Tracebility

As we use declarative model transformations, we can generate transformation traces that define which objects are transformed into which objects. This allows us to analyse how certain objects are created during the transformation chain, and in case of an error, it helps us to find the root of this error.

3.4 Integration of new platforms and devices

Because of the structure of the transformation chain, models can be reused and interchanged to generate a user interface for a different modality and implementation technology. To illustrate this, the concept of model reuse is depicted in Figure 3.4.

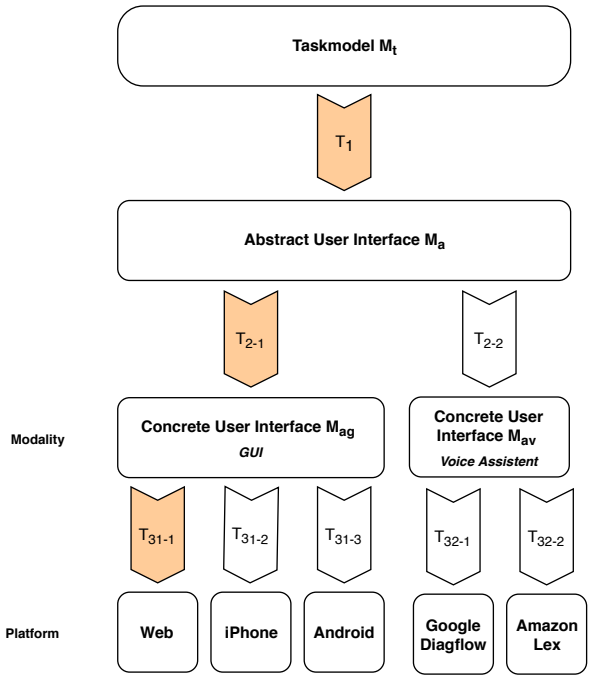


Figure 3.4: Models and transformations for different platforms and modalities

From the task model, we can generate with T_1 an abstract user interface. From the abstract user interface, we can transform the abstract objects to concrete objects. While the concrete objects are expressed for a certain modality, we can define for each modality a separate model and transformation. For example, we define a model transformation $T_{2\text{-}1}$ for a concrete user interface model of a graphical user interface M_{ag}. The same applies for a concrete user interface for a voice assistant.

Figure 3.5 illustrates the concepts involved in the task model and the abstract user interface model. For the purpose of explaining the possibility to reuse models of the transformation chain, we defined a simplified workflow called *Create a transaction*. This transaction requires at least the user to input an amount of *Money* and the *Receiver*. For each input variable, we defined a separate *Task* object. This is, however, not required if the input variables are not dependent on each other. The abstract user interface defines the structure and behaviour. The involved concepts allow the interaction to be structured in *Container* objects. These container objects contain the construct elements, such that the user can perform the task.

From the abstract user interface, we define different branches for different modalities. An example for a graphical user interface (GUI) is illustrated in Figure 3.6a and a voice assistant user interface (VUI) is illustrated in Figure 3.6b. Both models define an interaction with the user but use different concepts as interaction objects. In the GUI model, the user interface is defined as a Container that contains text fields for numerical and textual input, a button to confirm the input and a checkbox list to select an item. In the VUI model, the objects are limited to a dialog with questions and answers. We define dedicated metamodels for both modalities and defined these in Section 5.

3.5 Conclusion

The structure of the transformation chain allows the development of user interfaces with the same functionality for different platforms and modalities. Models can be reused and model transformations can be used to generate different branches for modalities and platforms. The building blocks of CbyC form the guidelines for the used methods and techniques to define the intermediate models and model transformations in the transformation chain. The conceptual model enables us to reason about the interface with the system.

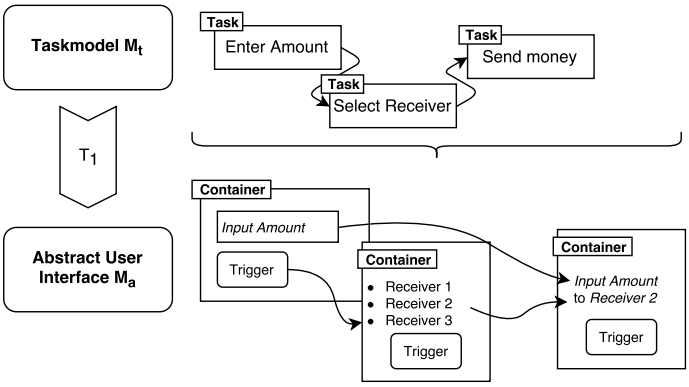
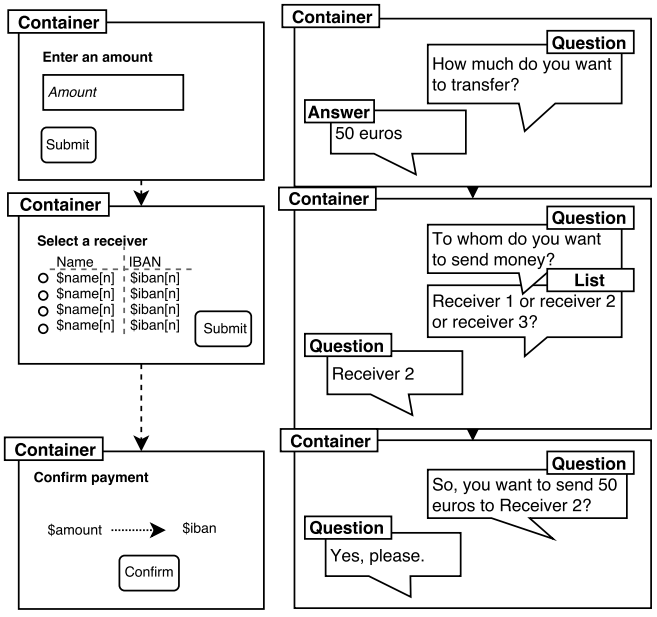


Figure 3.5: The workflow *Create a transaction* in the task model and the abstract user interface model



(a) Graphical user interface model (b) Voice assistent user interface model

Figure 3.6: Concrete user interface models for different modalities for *Create a transaction*

Chapter 4

Task model

In this chapter we describe the design choices for the task model with the design guidelines from the previous section. This model forms the input for the transformation chain.

4.1 Overview

The task model forms the highest level of abstraction in our transformation chain. It defines the workflow, as sequence of tasks, from a user perspective and can be used to analyse the interaction between user and system and as the source to generate the abstract user interface from. The goal of a task model specification is to define which tasks have to be performed to reach a certain (business) goal.

In our solution, business analysts with a technical background should be able to define the specification of the resulting user interface. Therefore, task modelling should be simple enough to understand and to be carried out. We also require a rigorous notation such that we can validate the specification at design-time and generate an operational user interface. Whereas such a notation involves complexity related to task ordering, interaction with system services and internal state management, the challenge is to balance the amount of complexity incorporated in the task model. In addition, we require the task model to fit in the MBUID process as we defined in Chapter 3. Consequently, the task model should not contain a large semantic gap with the other abstraction levels. As our goal is not to define a new approach to task modelling (since this does not fall within the scope of this research), we analysed the capabilities of the existing task modelling approaches. As a result, we defined a task modelling method that uses techniques from different approaches such that the complexity of the task model can be tackled step by step, and can be used as a source to generate an operational user interface.

In this chapter, we gradually define a rigorous specification of the task model. First, we define a *structural model* that uses temporal constructors to sequence tasks in workflows (Section 4.3). Second, we mark the structural model with an interaction specification to specify the communication with the user and the system (4.4). Lastly, from the marked structural model

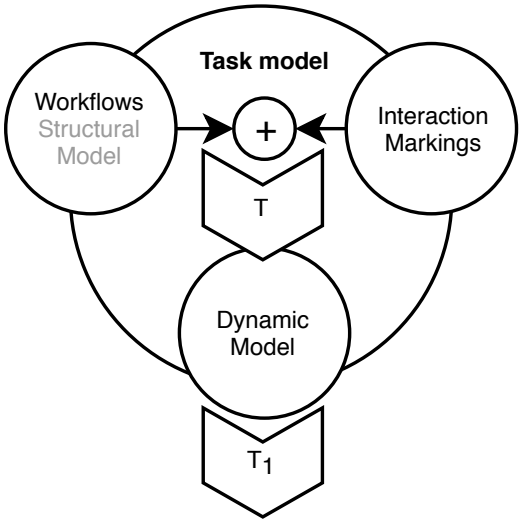


Figure 4.1: The tasks in the structural model annotated with an interaction class forms the input for the transformation to the dynamic model.

a model transformation generates a specification that defines *dynamic tasks* which specifies the input and output flow of data and events in the *dynamic model* (Section 4.5). The approach we used is depicted in Figure4.1.

4.2 Task modelling techniques

In the field of User-Centered Design (UCD), task modelling and analysis have been widely accepted as one fundamental way to ensure user-centered design [43] and to improve the understanding of how a user may interact with a user interface. A task model is often defined as a set of interactive tasks performed by either the user or the system, or by both, through the user interface. We discuss the main methods in task modelling and analysis involved in user interface development.

4.2.1 Hierarchical Task Analysis

Hierarchical Task Analysis (HTA) was among the first task analysis method that defined a task model in terms of tasks, task hierarchy and plans [44]. The primary goal of this method was to train users to perform certain tasks. Tasks are recursively defined by decomposing tasks in smaller subtasks until the task can not be decomposed any further. In that case, the task will be allocated to the user or the user interface. Since the tasks and subtasks do not define a specific order, plans define the order in which tasks have to be executed.

4.2.2 GOMS

The GOMS method describes and analyses how a user should perform their tasks in terms of Goals, Operators, Methods and Selection rules. Since the introduction, different forms of GOMS have been developed, each focusing on a different notation with different analysis goals. With the original GOMS model as the root of the family, KLM-GOMS [45], CMN-GOMS [45], Natural-GOMS-Language (NGOMSL) [46] and CPM-GOMS [47] are derived from this model. The Keystroke-Level Model (KLM) estimates the execution time for a task based on the actions the user must perform in terms of primitive operators. The CMN-GOMS defines a strict hierarchy of tasks with the goal to predict the operator sequence and the execution time. Natural-GOMS-Language (NGOMSL) defines a task model with task written in natural language with the goal to predict the the operator sequence, execution time, and time to learn the methods. Cognitive-Perceptual-Motor GOMS (CPM-GOMS), just as the other GOMS models, predicts the execution time. However, unlike the other models, the operators are defined as perceptual, cognitive and motor acts.

4.2.3 Groupware

Groupware Task analysis (GTA) is a technique to model the complexity of tasks in a cooperative environment [48]. Different to other task modelling methods, GTA focuses on people, work and the situation.

4.2.4 ConcurTaskTree

ConcurTaskTree (CTT) is a descriptive notation for defining task model specifications for interactive applications [49]. CTT has a formal definition of the temporal operators which originates from process algebra. The task model, defined as a recursive tree, consists of a root task decomposed of subtasks related with temporal operators. The definition of a task is defined as an action that manipulates an object. Tasks can be assigned to specific platforms to support different tasks for different target platforms.

4.2.5 MAD

MAD provides an object-oriented task modelling method which defines tasks, users, objects and constructors [50]. Tasks are decomposed in subtasks, constructors constrain the execution of the task, tasks are related to specific users and objects are manipulated in the task. Tasks are related by synchronisation operators (i.e., sequence, parallelism, and simultaneity), ordering operators (i.e., OR, AND, XOR), temporal operators (i.e, begin, end and duration), and auxiliary operators (i.e., elementary or unknown).

4.2.6 *Task Oriented Object Design*

Task Oriented Object Design (TOOD) defines tasks as objects with an input transition and an output transition [51]. A task can only be performed when the input conditions, such as the availability of data, are met. Tasks that are triggered by events can be decomposed in subtasks.

While HTA, GOMS and MAD are restricted to decomposing tasks into subtasks related by temporal operators, we consider these approaches not as expressive as Groupware, CTT and TOOD. While TOOD uses mathematical functions to define when tasks has to be performed, this approach tends to become complex. However, since TOOD can be simulated using Petrinets, strong validation can be achieved.
CTT on the other hand, is not as complex as TOOD, and has been widely recognised as a notation for task modelling. Existing MBUID approaches, such as AMBOSS [52], useML [53], taskMOD [54], UsiXML [15], MANTRA [55] and THERESA [56], are similar to CTT. This comes with the price that only the temporal constructors integrated in CTT are formalised using temporal algebra. The interaction with domain models is limited to defining access to attributes of domain models and if the user has to manipulate or to perceive the attribute. Whereas this approach is sufficient for a wide range of use cases, we define a more rigorous approach that defines the interaction that has to be carried out in a task. As a result, we are able to validate if the generated implementation enables the user or system to carry out this interaction.

4.3 Workflows in the Structural model

The structure of our task model is based on the ConcurTaskTrees (CTT) notation. As we discussed, CTT is a descriptive notation suitable for defining task model specifications for interactive applications[49]. Due to the well-defined hierarchical structure and expressive temporal constructors, this notation provides means to define and analyse task models. Using this notation we define a task model as a tree with nodes as tasks that can be performed by either the user or the system. Using the tree structure, each task can be divided into smaller sets of subtasks until the task cannot be decomposed any further. We call these non-decomposable tasks *interaction tasks*, as these tree leaves describe the actual interaction with the user interface, e.g., select an item or give an input. Branch nodes are used to create the necessary abstraction to define separated branches for specific tasks. We call these branch nodes *abstraction tasks*. Temporal constructors define the execution order. We define the set of temporal relationships in Table 4.1.

4.3.1 *Formal description*

The structural task model is structured as an recursive tree with a single distinguishable root node as the top node with the direction of the vertices going downwards. Each task can contain an ordered list of subtasks as children. Temporal constructors are used to define the possible execution paths between two subtasks at the same level. Tasks can only have a temporal constructor with the previous or next task in the list. The operational semantics of these constructors are defined in Table 4.1. The amount of subtasks of a task is not bound to any limit.

Name	Symbol	Example	Meaning
SequenceEnabling	>>	t1 >>t2	t1 has to be finished before the user can start t2.
Choice	[]	t1 [] t2	The user chooses explicitly between t1 and t2. The user can't execute both.
Interleaving	|||	t1 ||| t2	The user can start both tasks at the same time, or switch between tasks when a task is not finished yet.
OrderIndependence	| = |	t1 | − | t2	The user chooses explicitly between t1 and t2 and should execute both. The first task should be finished before starting with the next task.
SuspendResume	|>	t1 |>t2	t2 can cause t1 to suspend. When t2 finishes it can reinstate t1. For example, in this context t2 could be rendering an image such that the input fields in t1 can't be edited.
Disabling	[>	t1 [>t2	t2 can cause t1 to disable. When t2 finishes, t1 can't be reinstated.

Table 4.1: Temporal constructors with their symbols and meaning as defined by CTT [49]

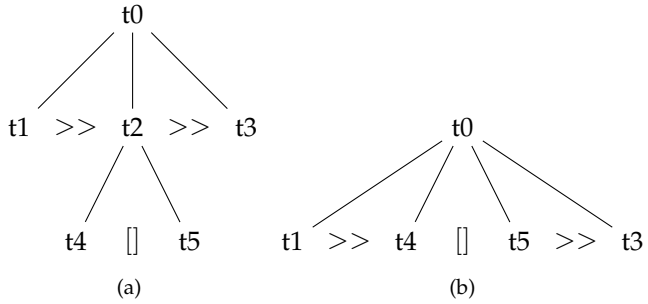


Figure 4.2: Example of two similar task models

4.3.2 *Operational Semantics*

The CTT notation allows one to define workflows of the task model succinctly. However, defining the temporal constructors that should define the order of tasks is not sufficient to obtain the supported workflows in all cases. For example, if we look at Figure 4.2b, it can be unclear which workflows are supported. Therefore, we need to define a solution for this problem. A typical solution would be to define precedence and associativity conventions among operators to resolve ambiguity [57]. However, we would argue that the hierarchical structure gives the user enough possibilities to define such a priority among operators by defining dedicated subtrees. In addition, this would increase the complexity of the task model and that is what we want to avoid. The simplest approach would be to define no precedence and left associativity among operators. In that case, the possible execution traces of the task model can be expressed in the following formula in Listing 5. The original approach of CTT defines a priority amongst operators, as defined in Listing 6, which results in a different formula, as listed in Listing 7.

$$t0 \iff t1 >> (t4[](t5 >> t3))$$
$$t0 \iff t1 \vee (t4 \wedge (t5 \vee t3))$$

Listing 5: Formula to evaluate if t0 is finished when operators are left-associative without precedence

$$\begin{aligned} Choice &\quad [\ ] \\ Interleaving &\quad ||| \\ Disabling &\quad [> \\ SuspendResume &\quad |> \\ OrderIndependence &\quad |=| \\ SequenceEnablingInfo &\quad [\ ]>> \\ SequenceEnabling &\quad >> \end{aligned}$$

Listing 6: Precedence amongst operators according to the CTT notation

$$t0 \iff t1 >> (t4[]t5) >> t3$$
$$t0 \iff t1 \wedge (t4 \vee t5) \wedge t3$$

Listing 7: Formula to evaluate if t0 is finished when operators are left-associative with precedence as defined in Listing 6

Both solutions are examined, and both solutions should resolve ambiguity. Further research is needed to verify if our concerns about increased complexity are legitimate. A different solution is to prohibit the use of different temporal constructors on the same level. For this solution we would require clear rules to define what combination of constructors are allowed. This solution however, would lower the expressiveness of the language, and the modeller is required to understand and apply these rules.

For the rest of this research we choose the simplest approach of defining no precedence and left-associativity among operators. When the supported workflows of the task model are not clear from its visual presentation, as in Figure 4.2b, the user can examine the generated execution traces to get the necessary feedback. In Appendix B, we defined a procedure to obtain these execution traces.

4.4 Interaction Specification Markings

The structural task model defines workflows that the user interface has to support. The intention of the task, however, is only captured in its name. Without a definition of the required interaction, we cannot generate an implementation. Therefore, interaction classes are defined which can be used to annotate tasks to further specify the interaction carried out in a specific task. The abstraction tasks in the structural task model decompose task into smaller non-decomposable interaction tasks. It is these interaction tasks which need to be annotated with a type of interaction. It is important to understand that the function of the abstraction tasks in the structural model is to decompose tasks in smaller subtasks. The leaves of the tree contain the actual interaction. We present an example, illustrated in Figure 4.3, to explain how these classes can be used to specify the interaction carried out in the tasks. In this model, the *Create a Transaction* workflows defines a process that enables the user to enter the required information for transferring money. Tasks labeled with a ($\rightleftharpoons$) symbol denote an interaction task, and thus these tasks need to be annotated. We discuss the different types of interaction that can be assigned to these tasks.

4.4.1 User Interaction

We define a user as a human actor capable of perceiving, processing and manipulating information. The user interface facilitates the perceiving and manipulating processes by respectively presenting the information appropriately and giving the user the right tools such that he finishes his task. To define this interaction with the user, user interaction classes define the bidirectional interaction between the user and the user interface. The classification of the interaction classes is based on the *ElementaryUseObjects* as defined as part of useML [58]. UseML defines 5 distinctive interaction

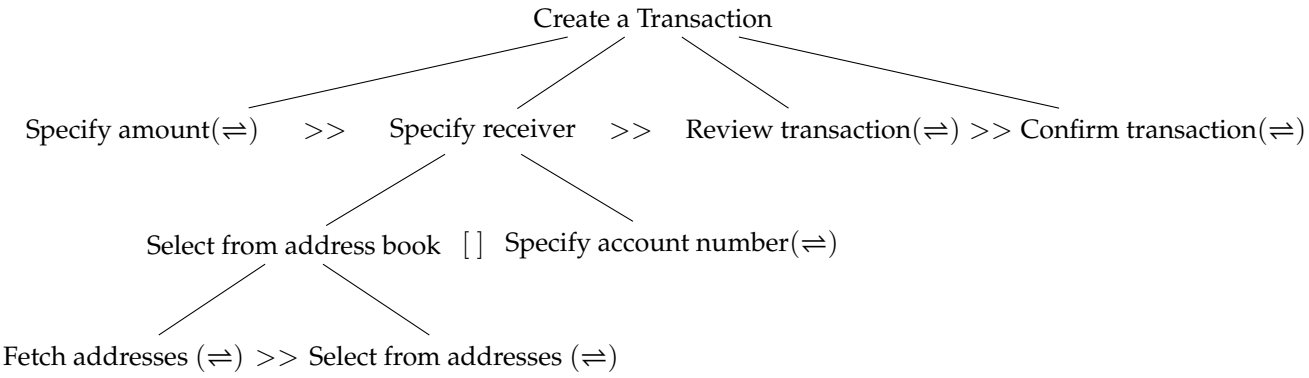


Figure 4.3: Example of the task model *Create a Transaction*

classes: execute, select, dataInput, change and inform. These classes define interactions of user tasks in a platform independent way and correspond to the definition of a user. We adopted different terms for these interaction classes, which correspond with the user perspective of the task model: trigger, select, generate, manipulate and observe, respectively.

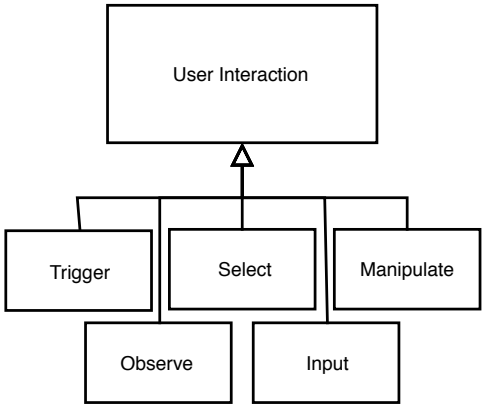


Figure 4.4: Overview of user interaction classes

To mark the tasks with the appropriate interaction class, insight into the type of input the system expects is required. For example, the task *Specify receiver*, in Figure 4.3, can be specified as a task that requires the user to select a receiver from a list, fetched by the system, or by entering an account number. Whereas both approaches result into a value that defines the receiver, the former requires a user to *select* an account, and the latter requires *generating* a value that correspond to the account number format. In another scenario, the receiver may be pre-defined and the user has to verify if the value is correct. In that case, the user may want to *manipulate* the value. The example in Figure 4.3, adopts the first two approaches and annotates the *Select from addresses* task with the *select* interaction class. The *generating* class is assigned to the *Specify account number* task. In the transformation chain both tasks will be implemented differently with different tools for the user to select a *receiver*.

Comparing the example in Figure 4.3 with the examples in previous sections, this example is more elaborated. The reason is that, besides user tasks, we also define system tasks in the task model. System tasks define the interaction with the system and therefore which data is available to the user. Without any information about the state of the system, carrying out the specified interaction would be cumbersome.

4.4.2 System Interaction

The conceptual model in Section 3.2 defines the total set of operations that an given actor can perform with the system, which comprises a subset of all operations. These capabilities consist of commands that invoke transitions and queries that retrieve information.

To specify the interaction with the system we define the input and output interface of each interaction. This way, we hide the complexity of system interaction. Queries tend to become complex when aggregations and complex filtering is necessary. In addition, queries can become dependent on each other when the response references other objects and additional queries are a necessity. Another problem with queries is that the structure of the response of queries does not always reflect the structure of construct elements in the user interface. The user interface should select elements from the response, possibly process the information, and map the values to construct elements.

We define two classes of system interaction: query and operation interaction classes.

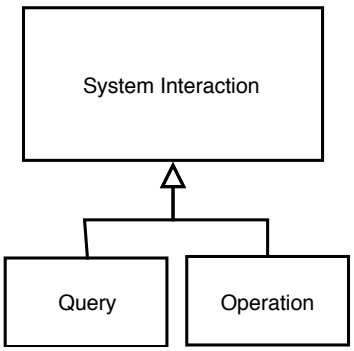


Figure 4.5: Overview of system interaction classes

Query interaction class

A query is a request for information. In our context, a query is defined as a request for information about *the state* of the system. Queries can request different kinds of data such as lists, (domain) object and specific values, depending on the requirements of such a query.

In the user interface, queries are used to obtain information such that the user can perform his task. The user interface should present this information to the user such that the user can make a well-informed decision. The task model does not define how information is presented, instead but it does define what information should be obtained from the system and presented to the user. Based on the conceptual model, the task model is able to query the attributes of an object and to query a list of objects with their attributes.

Command interaction class

Invoking a transition on a domain model can be specified using the command system interaction class. Invoking such a command requires the specification of the target transition, input data mapped to arguments of that transition and the identifier of the target object. In case the transition is a start transition, i.e., the object did not exist before the transition, the identifier of the target object is not required. Whereas we require the user interface to give feedback if the command is successfully invoked, such an interaction either answers with an error message as output or the output consist of information about the new state of the object as specified in the output transition.

The set of system interactions defined in the task model form the input for a transformation that implements the resolvers of these queries. A resolver calls the functions in the system to resolve the values of the response. When resolvers can not be generated automatically, due to the involved complexity, the implementation should be created by hand. This separate layer implements the queries in the task model, can be referenced throughout the transformation chain. This approach increases the flexibility, maintainability and hides the complexity of retrieving information about the state of the system.

These interaction classes allow us to mark the required tasks in the structural model. This marked model contains the workflows and the interaction with the user and the system. Where temporal relationships define the ordering between tasks, no such relation between tasks concerning the data flow is defined. For example, what is the purpose of a selected item? If data should be observed, which task does produce this information? In addition, temporal relationships do not define what happens when the user cancels the task. To adopt the notion of data flow and to support advanced scenarios, outside of the defined workflows, a dynamic model is defined to capture these semantics.

4.5 Dynamic Model

To specify the data flow and advanced workflows, we adopt the Task Oriented Object Design [51] in the dynamic task model, to specify tasks as objects with an input and output transition. A dynamic task model defines data flows, system state checks, and events that trigger tasks.

4.5.1 Task object

In the dynamic task model we define a task as an object which consists of an input and output interface, and a body. The input interface defines the required input data necessary for the execution of the task. This data is considered as the initial condition to be able to execute the task. Three types of input conditions are defined: events, preconditions and input data. The body describes the subtasks and the data flow between subtasks and from the task to the subtasks. The output interface specifies the data that is created, the postconditions that must be met and which reactions are triggered.

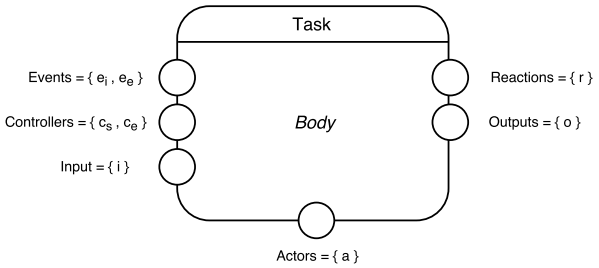


Figure 4.6: Task object with the elements of the input and output interface

Input interface

- **Triggers** To trigger the execution of a task, an event within the set of triggers should be raised either as a result of the previous tasks, or by the system. We define two types of events: internal events and external events. Internal events are produced by other tasks in the task model and are used to trigger the other tasks in the workflow. External events can be set off by the system as a result of an asynchronous call or to interrupt the current workflow such that the user should perform a task with a higher priority.

- **Input data** specifies the incoming data from another task to perform the task.

- **Preconditions** can be defined to constrain the valid input data to start the execution of the task.

Output interface

- **Reactions** can be produced by the user during the task and can be used as *Events* as input for other tasks.
- **Output data** specifies the data that is produced or transformed during the performance of the task and can be used by other tasks. By defining input/output sequences we can define a data flow from one task to another.
- **Postconditions** define the constraints of the values of the output data.

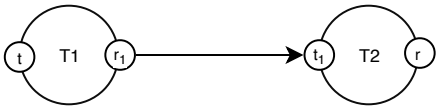
4.5.2 Event-Driven Tasks

Temporal constructors are capable of defining relationships between tasks. This defines the basic workflows that a user interface should support. Whereas these constructors are powerful, the specified behaviour is not completely deterministic. It does not specify what happens when tasks become obsolete, when users want to abort a task or go back to the previous task. In all cases, the user should be able to abort the task, and the user interface should return to a state that is relevant to the user. Since the Task Object Oriented Design defines the tasks as event-driven, tasks are triggered by the corresponding events. Tasks defined in the structural model can be transformed to event-driven tasks as defined in Figure 4.7. Complementary to the derived tasks, dedicated events can be defined that can be raised by the user to go back to the previous task or to abort the task.

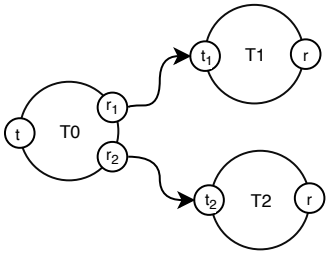
4.5.3 Input/output mapping interaction classes

Since every interaction task is annotated with an interaction class, this annotation specifies the type of input and output data. For example, a task which is annotated with the *select* interaction class, should receive a *list* as input and an *item* of that list as output. In the dynamic model we specify the constraints on the input and output of a task. For user interaction the input/output pattern is used in the transformation to select a construct element that is able to receive and provide the data. For the system interaction, the input/output pattern is used to generate the specification of the interface. We define the input/output pattern resulted from an interaction class as functions. We define the constraints on the input and output pattern in of both the user interaction classes and the system interaction classes in Listing 8 and the syntax is introduced in Listing 9.

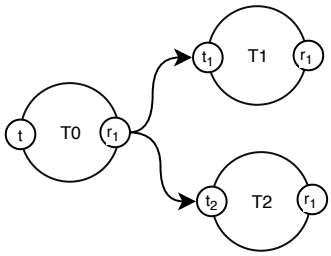
By defining tasks with input/output mappings, user interaction is abstracted from modality and platform. The interaction specification does not define how actors should carry out their tasks but defines the relationship between the input and output. In the transformation chain, model trans-



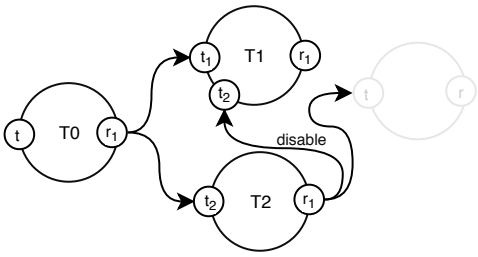
(a) SequenceEnabling / SequenceEnablingInfo



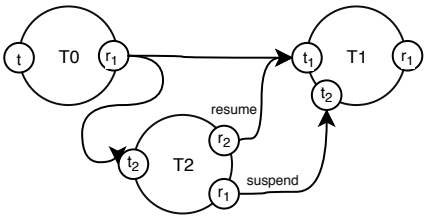
(b) Choice



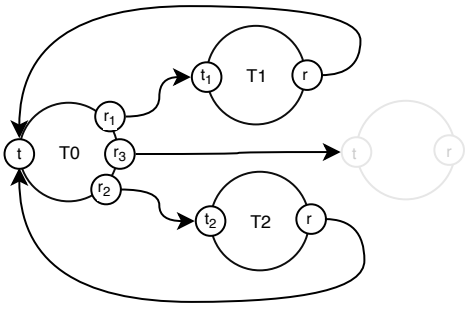
(c) Interleaving



(d) Disabling



(e) SuspendResume



(f) OrderIndependence

Figure 4.7: Temporal constructors expressed in event-driven tasks

$$\{i : Set < Item >\}\ Select\ \{o : Item\ of\ i\}$$
$$\{i : Set < Item >\}\ MultipleSelect\ \{o : Set < Item >\ subset\ of\ i\}$$
$$\{i : -\}\ Generate\ \{o : Value\}$$
$$\{i : Value\}\ Manipulate\ \{o : Value\}$$
$$\{i : Value\}\ Observe\ \{o : -\}$$

$$\{t : Type, id : ID\}\ GetObject\ \{o : Objectoftypet\}$$
$$\{t : Type, filters : Set < KeyValue >\}\ GetObjects\ \{o : Set < Object >\}$$
$$\{i : Set < KeyValue >, id : ID\}\ InvokeCommand\ \{o : Value\}$$

Listing 8: User Interaction classes projected as functions

formations use this specification to derive the corresponding construct elements and create data bindings to retrieve and store the corresponding data. In the dynamic model, this input and output interface specification allows the validation to enforce that the specified interaction can be carried out due to the availability of input data.

$$\{I\}C\{O\}$$
$$I = Input\ Constraints$$
$$C = Name\ of\ interaction\ class$$
$$O = Output\ Constraints$$

$$DomainModelTypes = Set\ of\ types\ of\ Domain\ Models$$
$$Type = member\ of\ DomainModelTypes$$
$$ID = Identifier$$
$$ValueSet = \{String, Integer, Double\}$$
$$Value = member\ of\ ValueSet$$
$$Object = instance\ of\ a\ Domain\ Model\ object$$
$$KeyValue = Key/Value\ mapping$$
$$Set < Item > = Set\ of\ items\ of\ type\ Item$$

Listing 9: Syntax of Listing 8

4.6 Implementation

To be able to specify an instance of a model we defined a metamodel of the task model that comprises the features of the structural model, dynamic model and the interaction classes. The metamodel defines the involved concepts and the relations between concepts in an instance of that model.

The metamodel is illustrated in Appendix C. The *Task* class forms the center of the metamodel and can be annotated with the *Interaction* class by defining an *interaction* relation. The enumarator, *UserInteractionType* and *SystemInteractionType*, defines the type of interaction in the Interaction class. The input and output of the interaction is specified as an input and output relation with a *Data* class. In this class, the data can be named, given an unique identifier, a data type, and a label. Implementing the event-driven tasks requires an *Event* class and a relationship between the *Task* and *Event* class. The *Event* object specifies an event that either triggers tasks or can be produced by tasks. Temporal constructors can be defined by relating tasks with a *Relationship* object. To be able to refer to external task models, we defined *ProxyTaskDModel* objects.

4.7 Limitations & Constraints

Validation at design time Whereas we enable the business analyst to define a task model by defining the structure and verifying if the task model supports the desired workflows, we can not validate at that point if this task model is valid. We can not check if the data complies with the input transition of a task.

Guaranteeing system output In addition, we define an interface with the system by defining only the external properties: the input and output. However, when the interaction does not involve a predefined query or command, we can not asses the validity of that system interaction at design-time.

Rigid relations The temporal constructors define a rigid definition of the relation between task. To enable custom paths, we defined event-driven tasks that can be triggered by tasks and raise events that trigger other tasks. Whereas we can derive event-driven tasks from the temporal constructors, the CTT notation is not valid when custom paths are added. Therefore, the business analyst should either understand the event-driven task structure, or feedback is required.

Statically defined input/output The current definition of a task is based on static data; the type of data and the value is defined at design-time. However, in some cases, which data objects will be created may be unknown when for example, the user has to enter an arbitrary amount of metadata.

4.8 Conclusion

The task model is defined as such that different projections can be derived. The structure, data flow and the dynamics can be modelled in separate projections, tackling the complexity in steps, enabling business analysts with a technical background to gradually define and refine the task model. The structural projection defines tasks, decomposed in smaller tasks related with formalised temporal constructors from a widely accepted technique, CTT. However, in CTT the interaction with the user is limited to defining which attributes of domain models should be perceived or manipulated. Our process of defining interactions makes the definition of this interaction more concrete, with the user and the system. We introduced the notion of system tasks to explicitly define what is expected from the system such that we can define the output of these system tasks as input for the user interaction. Whereas the interaction description relates the input to the output, we can use this input and output description to define the data flow between tasks and validate if data will be available for the use to perform the defined interaction. The data flow enables the modeller to specify where the generated data, by the user or the system, will be used as input for the user or system tasks.

Chapter 5

User Interface Models

In this section we address the design choices involved in defining the intermediate user interface models. Both the underlying concepts as well as the implementation in a metamodel for both models will be discussed.

5.1 Purpose

The task model defines the tasks that should be supported by the User Interface to interact with the user and the system. As we want to transform this specification to an operational user interface, we define two models that allows the gradually refinement of these models for a specific modality and platform. The goal of these models is to express the structure and logic of a user interface in such a way that transformations can transform this to different modalities and platforms. It is important that the models should preserve the operational semantics of the task model; both models should contain the same workflows as the task model, and the input and output references should be preserved.

In this chapter we define two user interface models that resemble in structure: the abstract and concrete user interface. The former defines a representation of the user interface abstracted from platform and modality and refines the task model by adding user interface specific constructs and logic. The latter refines the abstract model by transforming the specified constructs in concrete constructs with presentation style and platform specific behaviour. In this chapter we discuss the concrete user interface model for a graphical user interface.

5.2 Abstract User Interface

In the abstract user interface we define construct elements, API calls and user interface behaviour. Similar to UsiXML, we use events and event listeners to model the behaviour of construct elements. Events are raised by construct elements to denote an action of a user and raised by function calls to denote the result of an interaction with the system. This approach is illustrated in Figure 5.1.

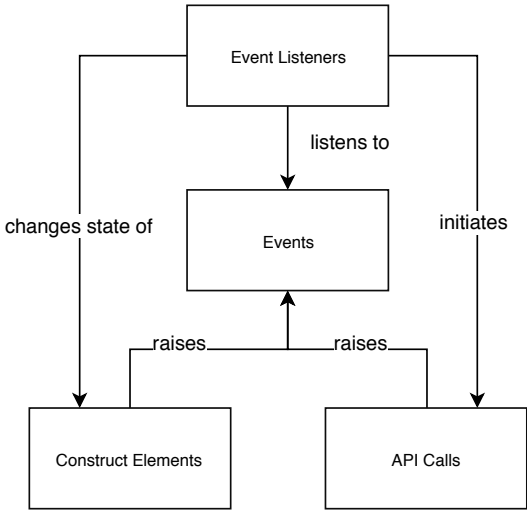


Figure 5.1: Conceptual overview of defining behaviour of the abstract user interface with *EventListeners*

5.2.1 Abstract Construct Elements

Containers, Triggers, Lists, Input and Output objects form the elementary objects of this model and provide the structure and the tools such that the user can perform the specified tasks. We call these objects, AbstractUserInteractors (AUIs). A *Container* groups a distinct set of AUIs such that the accessibility of these elements can be toggled. A *Trigger* defines an atomic action of the user to, for example, confirm, reset, or start a task. A *List* enables the user to perceive a structured set of information and to select a single or multiple items of that list. An *Input* object enables the user to input or to manipulate information. An *Output* object specifies that information has be communicated to the user.

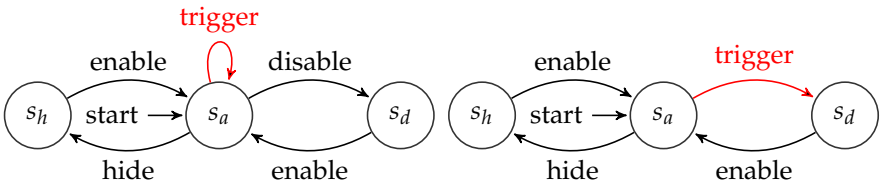
The concept of abstract construct elements and their characteristics is easier to grasp if we project these elements on a specific platform and modality. For example, a typical GUI consists of a form that enables the user to input and to manipulate data. An instance of an abstract user interface that models such a form can be defined as a Container that contains Input, Output, List and Trigger elements. When we project this structure onto a GUI, the Input elements will be transformed into a text field, an integer field or another field that enables the user to provide information. Output elements will be transformed into text, a graphic or different form of output. Lists will be transformed into dropdown elements that enables the user to choose a value. Triggers will be transformed into buttons such that the user can confirm or reset the information.

5.2.2 *Events*

We define two types of events: internal and external events. Internal events denote a state change of a construct element. External events denote the result of an API call.

Internal events

The different states and transitions of an abstract construct element can be defined in an abstract state machine*. Such a state machine denotes which transitions can be triggered in which state. Both the user as well as the user interface can trigger these transitions. An event is raised when the state of an AUI element changes. The behaviour of each AUI is defined in a separate state machine. Therefore, we can customise the behaviour of the same AUI with a different behaviour. An example of two state machines for the same AUI is illustrated in Figure 5.2.



(a) trigger does not change the state

(b) trigger changes the state into s_d

Figure 5.2: Two different state machines of the abstract construct element *Trigger*

The state machine in Figure 5.2a describes the behaviour of a Trigger element with 3 different states: *Hidden* (s_h), *Accessible* (s_a) and *Disabled* (s_d). Only in the *Accessible* state, the user can trigger the *Trigger* event. Figure 5.2b defines the same Trigger element, but when the user triggers the *trigger transition* , the state of the element will change into the state *disabled*. In this case, the user can only trigger the Trigger once. This can be usefull when a user has to confirm information. Another use-case is to change the start state in, for example, the disabled state. The user interface can communicate to the user that the possibility exists but the user can not trigger the Trigger, yet.

Using this separate state machine we can customise the behaviour of the AUI elements and express design choices at this level in the transformation chain.

*Not to be confused with ASM's as a method for the design and analysis of complex systems

External events

In our approach we define system calls asynchronous by definition. As a result, an external event denotes the start and finish of a function call. When the function call returns a result, an event is raised. When the function call is not successful, the user interface can react on this event by showing feedback, or by aborting the task.

5.2.3 Event Listener

The event listener approach is used to define inter-element behaviour. The event listener as a concept listens to events and can act upon events by changing the state of construct elements and send an request to the system. The collection of event listeners forms the orchestrator of the user interface. It defines the behaviour of the user interface based on transitions of AUI elements and the result of function calls.

5.3 Concrete User Interface

The CUI refines the AUI by adding information on how the UI has to be perceived and manipulated by the user. This model adds the notion of modality and is, therefore, modality dependent. The target modality determines the structure of the model and also the available construct elements that a user interface can support. For each modality we define a dedicated concrete user interface model that models the structure and behaviour of a user interface for that specific modality. We discuss the structure of an *graphical user interface* which resembles the structure of the abstract user interface model.

5.3.1 Concrete Construct Elements

The transformation of an abstract construct element to an concrete construct element depends on the context in which the element appears. For example, a list in a graphical user interface could a dropdown input or a simple checkbox list in a form, or the list could be displayed as a table as in a CRUD interface. Semantically these three options are the same, they each offer the possibility to select an item from the list, and in the abstract user interface they are modelled using the same object. In the concrete user interface these elements are refined such that they fit in their context.

The refinement of these elements focuses on the corresponding modality concept and the styling. Both are modality dependent. The modality concept determines if the list is a table or dropdown, and the styling specifies how the concept should be presented to the user.

Modality Concepts

For a graphical user interface a set of modality concepts is defined that is supported by a wide-range of devices that offers a graphical user interface. For a graphical user interface we identified distinct classes of concepts that are supported on *touch* devices and *mouse-keyboard* devices.

Concrete state machines

In the abstract state machine, each transition is defined by the source and target state. In the concrete state machine we expand this event definition by means of a concrete interaction. For example, triggering a trigger could be by means of a click or a touch, or both. By linking these interactions with transitions, we can customise these state machines such that they fit in a particular context. Another example comes up when you consider a target device with limited capacity, for example, in case of a mobile device. In that case a separate state machine can define the behaviour of a collapsible menu such that the container changes in size when the user triggers it. To model this behaviour we define the visible state as a compound state in which the state of the container can either be closed (but visible) or open. Figure 5.3 illustrates a concrete state machine for a collapsible container.

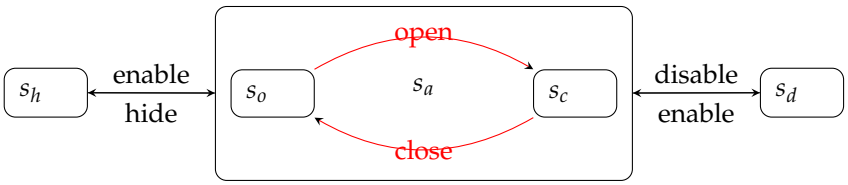


Figure 5.3: Example of a use-case for a concrete state machine to describe the behaviour of a concrete *collapsible* construct element

Building blocks

When this state machine has been defined, this could be used for different situations without redefining the same behaviour. This results in a library of concrete user interface elements that can be combined when needed.

Styling

We add styling to a construct element by mapping states to style classes. If a concrete construct element only has three states, the styling of each state corresponds to the characteristic of the state. For example, a button in the disabled state can be greyed out and hidden in the hidden state. The majority of effort in styling is involved in the *accessible* state. Depending on the modality element, different styling attributes are relevant.

5.4 Metamodels

We define metamodels for the abstract and concrete user interface and state machine.

5.4.1 Abstract User Interface

The structure of the metamodel of the abstract user interface resembles the structure defined in Section 5.2. The metamodel defines the *AbstractUserInteractor* as a class that is contained by the *AbstractUserInterface* which contains the events, data and construct elements. The *Data* class is related to the AbstractUserInteractor class and the *SystemInteraction* class. The SystemInteraction class defines the function calls that either query or invoke a command on the system. The *EventListener* class defines the behaviour when either an AbstractUserInteractor or a SystemInteraction object raises an *InternalEvent* or an *ExternalEvent*. The *Condition* object defines a condition when the reaction should be triggered. The AbstractUserInteractor forms the supertype of the abstract construct elements which we defined in Section 5.2.1. For each element, a class in the metamodel is defined.

5.4.2 Concrete Graphical User Interface metamodel

We defined a concrete user interface model for a graphical user interface. The general structure of a userInteractor, System Interaction and the EventListener remains intact as these concepts are still valid in a graphical user interface. The metamodel of the concrete user interface can be found in Appendix D. What differs is that the AbstractUserInteractor is refined into a *ConcreteUserInteractor* which is a supertype of the modality concepts that are supported by the target modality. The set of construct elements which we defined is depicted in Figure 5.6. Each ConcreteUserInteractor can be mapped to a *ConcreteStyleModel*. The metamodel of the *ConcreteStyle* model is depicted in Figure 5.4. In this model, each state can be mapped to a *ConcreteGraphicalStyle*. This class does not contain an exhaustive set of styling attributes, but it does show how styling can be defined.

5.4.3 State machine

We defined a metamodel for a state machine that defines the behaviour of the *AbstractUserInteractor* and *ConcreteUserInteractor*. For the *Statemachine*, *Transition* and *AbstractState* object, the concrete state machine can refer to an abstract user interface via the *conformsTo* relation.

5.5 Conclusion

The defined metamodels define concepts that can be used to model the structure and logic of a user interface. In addition to current approaches, we defined an extra class, *SystemInteraction* such that system interaction can be specified. An alternative approach is to refer to *SystemInteraction* objects in the task model. In that approach we avoid the repetition of the same information. However, when the task model changes, or iterations are being developed, the references are likely to be broken, or do not comply with the structure and objects of the abstract and concrete user interface models. We choose for internal consistency; the model remains consistent while other models may change. Changes in other models can be propagated when necessary, and do not directly corrupt other models in the transformation chain. We added state machines that enable the user to specify and customise the behaviour of the construct elements. This behaviour, together with the event listeners that define how and when state changes of construct elements should be triggered, can be used to check if state changes can be triggered. For example, when an event listener specifies that the button should be triggered, the element should be accessible for the user. We have to verify if we can statically analyse if the specified behaviour is consistent. In addition, we have to validate this approach to verify use-cases for the use of these state machines.

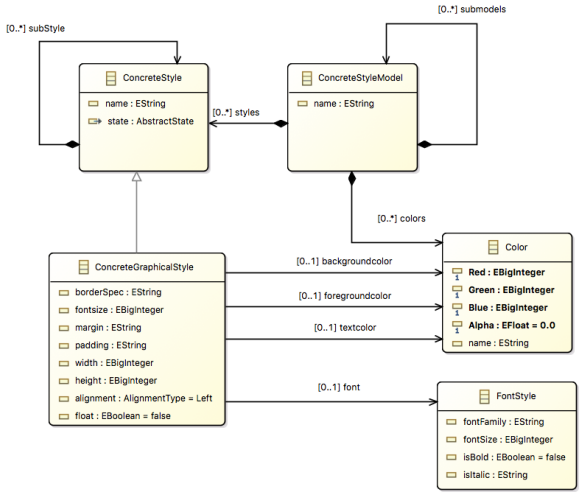


Figure 5.4: Metamodel of the Style model

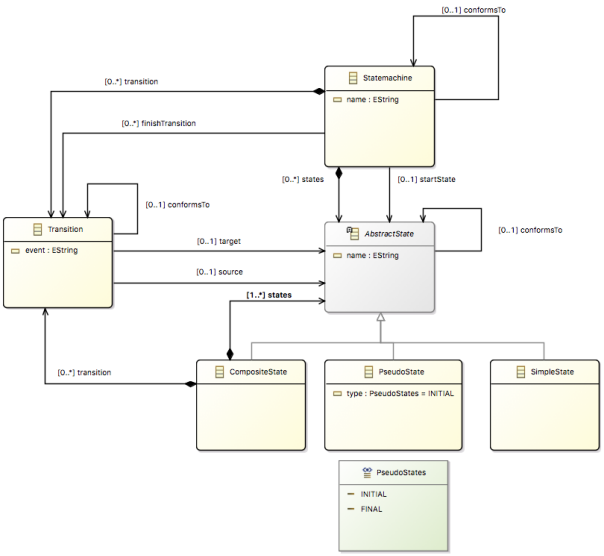


Figure 5.5: Metamodel of the Statemachine

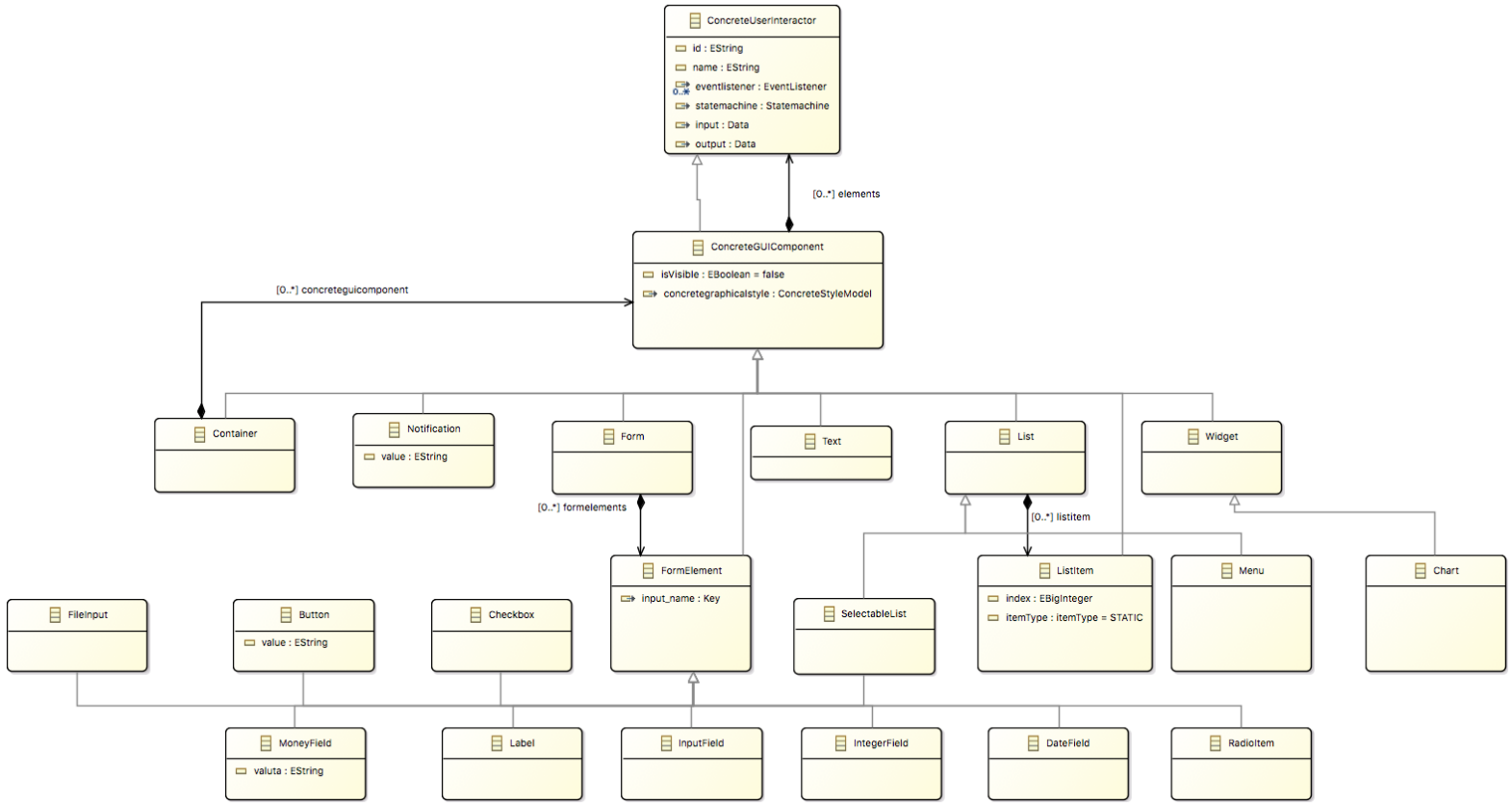


Figure 5.6: Metamodel of the Concrete User Interface model

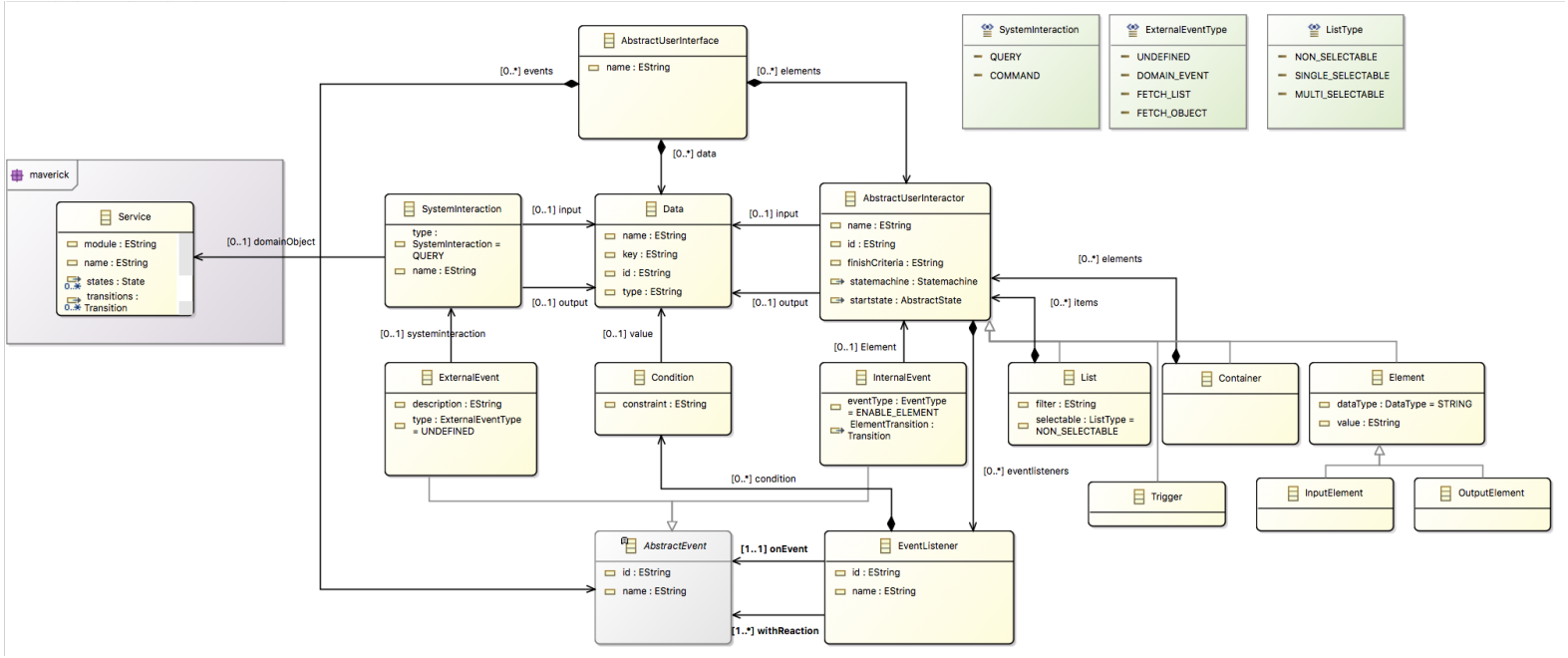


Figure 5.7: Metamodel of the Abstract User Interface model

Chapter 6

Model Transformations

In the previous sections we discussed the involved concepts of different models in the transformation chain. The goal of model transformations is to refine the objects of a model. We discuss the general structure of the transformation rules and the problems involved.

6.1 Transformation languages

To define a model transformation we need a language to encapsulate how concepts from one model transform to concepts from another model. In Table 6.1, an non-exhaustive overview of model transformation languages is presented with the use of evaluation properties defined by Czarnecki and Helsen [59]. The list of model languages are supported by tools and a large body of knowledge is available in literature.

name	type	tracebility	directional	tool-support
AGG [60]	graph	user specified	bidirectional	AGG Tool
ATL [61]	hybrid	automatic	unidirectional	EMF
ETL [62]	hybrid	user specified	unidirectional	EMF
GROOVE [63]	graph	-	unidirectional	GROOVE simulator
Henshin [64]	graph	automatic	bidirectional	EMF
QVT-Operational [26]	imperative	automatic	unidirectional	EMF
QVT-Relations	declarative	automatic	bidirectional	EMF
VIATRA2 [65]	graph	user specified	unidirectional	EMF

Table 6.1: An overview of model transformation languages

Among others, Query/View/Transformation (QVT) and Atlas Transformation Language (ATL) [61] are languages standardised by the Object Management Group (OMG) and well-supported with tools. Due to the declarative nature of ATL, the possibility to add imperative statements, and due to the large body of knowledge about the semantics and verification techniques, ATL is chosen to as the language for the model transformation definitions.

6.2 Transformation definitions

Since we require that we can automate the transformation from task model to an operational user interface, we define model transformations that facilitate this process. In this research, the purpose of the defined model transformations is to demonstrate that it is possible to refine models such that they result in an executable definition of a front-end application. We first discuss the general approach in the defined transformation. Then we discuss how this approach translates to the definition of rules.

6.2.1 Taskmodel to Abstract User Interface

Transforming the task model to an abstract user interface model consists of expressing relations among tasks and defining abstract construct elements such that the interaction can be carried out by the user. This first model transformation requires little to none customisation as the transformation from task model to abstract user interface can be defined in a straightforward approach.

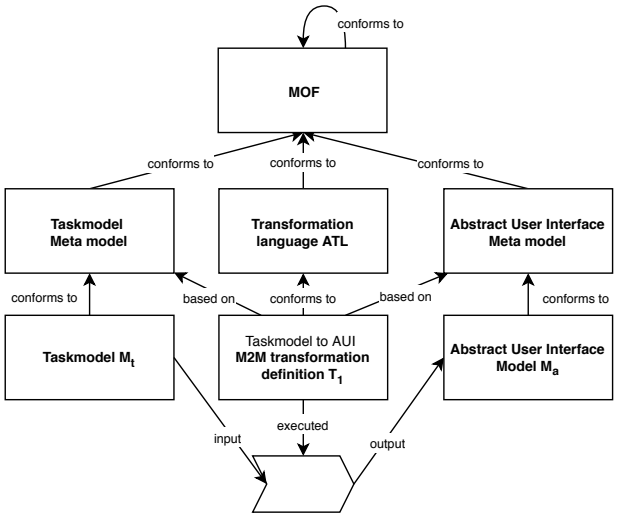


Figure 6.1: Conceptual overview of the first model transformation T_1

Transforming Relationships to Event listeners

To define user interface behaviour we create for each relationship the necessary event listeners. For each type of temporal constructor we define a pattern that implements the event listeners such that the definition of the temporal constructor is enforced. For each temporal relations a pattern is defined that translates the characteristics of the temporal constructor with events and event listeners.

```
rule getSequenceEnablingRightEventListener {
    from
        t: Taskmodel!Relationship
        (
            t.nature = #SequenceEnabling or t.nature =
 ↪  #SequenceEnablingInfo
        )
    to
        a: AUI!EventListener (
            onEvent <- l,
            withReaction <- r,
            withReaction <- l_hide,
            transitionType <- #NAVIGATION
        )
        l: AUI!InternalEvent (
            Element <- t.left_sibling,
            eventType <- #FINAL_EVENT
        ),
        r: AUI!InternalEvent (
            Element <- t.right_sibling,
            eventType <- #ENABLE_ELEMENT
        ),
        l_hide: AUI!InternalEvent (
            Element <- t.left_sibling,
            eventType <- #DISABLE_ELEMENT
        )
}
```

Listing 10: Excerpt from transformation TaskDModeltoAUI that defines a transformation rule for the creation of an event listener for the SequenceEnabling relationship

Finish criteria

Even though temporal constructors define the temporal relation between tasks, it only specifies the next tasks to be executed after finishing that particular task. It does not define when the task is considered as *finished*. Especially when a task is composed of subtasks, it is not straightforward to determine when a task can be considered as finished. Considering that a task contains either an interaction class or an arbitrary amount of subtasks, we define a strategy to resolve the finish criteria. To solve this problem, a helper is defined that traverses the subtree of that particular task and returns the finish criteria by means of a logical expression. To illustrate this, Listing 11 defines the formula that evaluates if the task *t1* defined in Figure 6.2 is considered as finished. The logical expression is defined using the semantics as described in Table 4.1.

$$(t3 \wedge \neg(t5 \vee t6)) \vee (\neg t3 \wedge (t5 \vee t6)) \iff t1$$

Listing 11: Formula that evaluates if t1 is considered as finished

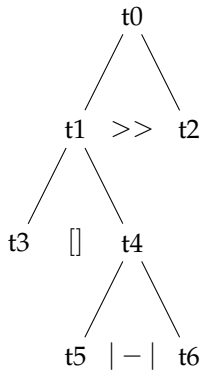


Figure 6.2: Sample task tree to illustrate the finish criteria of t1

According to the formula defined in Listing 11, *t1* is finished if either *t3* is finished or either *t5* or *t6* is finished. To capture this behaviour in the user interface, an EventListener object is defined. This EventListener object defines that when *t1* is finished, *t2* should be enabled. In the example of Listing 11 this EventListener object should listen to the finish criteria of *t3*, *t5* and *t6*.

6.2.2 *Abstract to Concrete User Interface*

The transformation from abstract user interface to concrete user interface involves the transformation from abstract to concrete construct elements. For an abstract construct element, multiple options can be considered in the concrete model. Therefore, we consider the mapping of these elements as a design choice. To express these design choices we defined a mapping model that maps instances of abstract construct elements to a concrete construct element type.

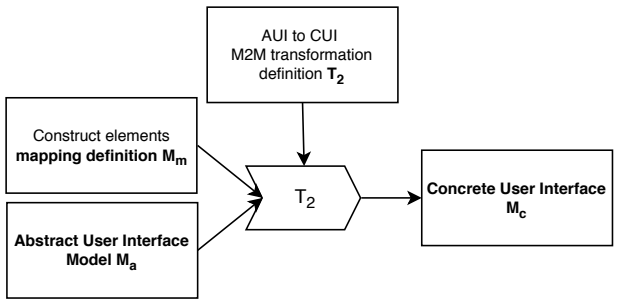


Figure 6.3: Input and output models of transformation T_2

The metamodel of the mapping model is depicted in Figure 6.4. The *Mapping* class maps an *AbstractUserInteractor* to a type of *ConcretaUserInteractor*. The supported types of the concrete model are defined in an enumerator *CUI_type*. In the transformation definition each construct element is either transformed according to the specified mapping, or the default mapping is

used. Because of this default mapping, the design choices only have to be expressed when the default mapping is not desirable.

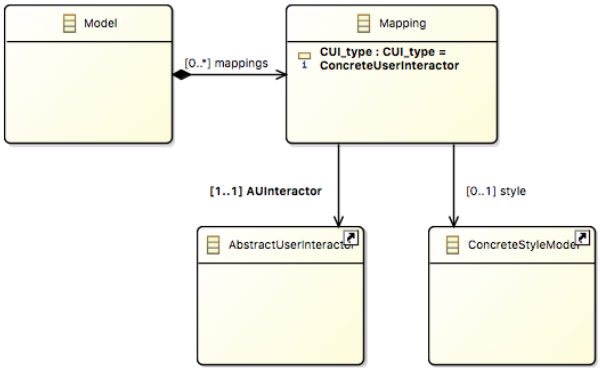


Figure 6.4: Metamodel of the mapping model

Using the mapping model, design choices are captured in a separate model. An alternative approach is to generate an instance of the target model and add design choices by hand. However, these design choices are not stored. When the source model changes, these design choices can not be reapplied and the model has to be edited by hand, again. By capturing design choices in a separate configuration file, the target model can be regenerated and it includes the design choices if the model is not changed significantly.

To use the mapping model in the transformation definition, the mapping model is considered as input. At least two matching rules are defined for each AUI element. One rule for the default transformation and one rule for a custom transformation, when the mapping model defines a mapping for that element. Helper functions are defined to verify in the matching rules if an mapping is defined for that element. Listing 12 depicts an example of a transformation definition of an *OutputElement*. When the element is not in the mapping model, the matching rule *toText* will be executed. Otherwise, the *OutputElement* will be transformed to a *Notification* element with the styling specified in the mapping model. The defined mapping rules are defined in Table 6.2. The default mappings is denoted with an asterisk.

6.2.3 Concrete User Interface to Final User Interface

The last transformation is strongly dependent on the target modality and platform. Therefore, the transformation we defined is very specific to the implementation technology. In this research, we choose a Javascript framework as the implementation technology for the front-end application. But in theory, every implementation technology can be used, as long as it supports the concrete construct elements defined in the concrete user interface model.

```
helper context AUI!AbstractUserInteractor def : inMapping(cui :
 ↪  MAP!CUI_type) : Boolean =
    MAP!Model.allInstances().first().mappings->
    exists(m | m.CUI_type = cui and m.AUInteractor = self);

helper context AUI!AbstractUserInteractor def : getMapping(CUI_type :
 ↪  MAP!CUI_type) : MAP!Mapping =
  MAP!Model.allInstances().first().mappings->
    select(a | a.AUInteractor = self and a.CUI_type = CUI_type).first();


rule toText {
    from
        a : AUI!OutputElement (
            a.notInMapping()
        )
    to
        c : CUI!Text (
            name <- a.name.toNamespace(),
            value <- a.name.toNamespace()
        )
}

rule toNotication {
    from
        a : AUI!OutputElement (
            a.inMapping(#Notification)
        )
    using {
        m : MAP!Mapping = a.getMapping(#Notification);
    }
    to
        c : CUI!Notification (
            value <- a.value,
            concretegraphicalstyle <- m.style
        )
}
```

Listing 12: Excerpt from the transformation AUItoGUI to illustrate how mappings are integrated in matching rules

Because the Javascript framework composes the user interface as a tree with nodes as components, the transformation is straightforward: every *Container* can be modelled as a component and an HTML component is created for each construct element. More details on the implementation can be found in Appendix F.

abstract	rule (in AUItoGUI.atl)	concrete
List	*reserved*	List*
	toDynamicDropdown	SelectableList
	toMenu	Menu
Input	toTextField	TextField*
	toMoneyField	MoneyField
	reserved	FileInput
	toIntegerField	IntegerField
	toDateField	DateField
Output	toText	Text*
	reserved	Chart
	reserved	Widget
	toNotication	Notification
Trigger	toButton	Button*
Container	toContainer	Container*
	toMenu	Menu
	toForm	Form

Table 6.2: Mapping of abstract construct elements to concrete construct elements

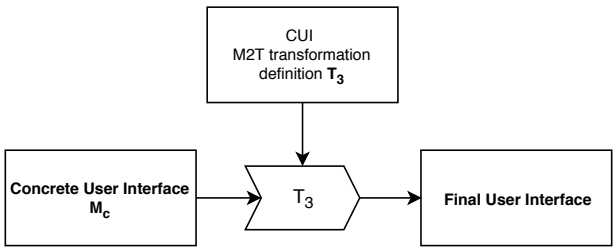


Figure 6.5: Involved models in T_3

6.3 Conclusion

In this section we defined an approach to implement the model transformation that transforms well-formed instances of models in the transformation chain to more refined models. Since we want to automatically generate an implementation, the complexity of translating a model definition to source code is shifted to the transformation definition. In practise defining a correct model transformation costs a significant amount of effort as defining declarative model transformation concentrates the complexity on a limited amount of matching rules. The advantage is that a such an transformation is deterministic and therefore predictable. As a result, we bootstrapped the transformations to a limited set of temporal relationships to demonstrate the transformation chain. To generate more complex user interfaces, we need to invest more effort into these model transformations.

Chapter 7

Leveraging Domain Models

In the previous sections the structure and concepts were defined that form the models of the transformation chain. In this section we define how a behavioural domain model can be used as input to our transformation chain.

In the task model we are able to refer to characteristics of domain objects in the system: tasks can query the state of objects and invoke commands to change their state. Instead of defining the specification by hand and referring to elements in the domain models, the reverse approach would be to generate a specification from a domain model specification and customise these by hand. We call this approach the boilerplate approach.

7.1 Generating task model specification

If we recall the behaviour specification of an object as we specified in the conceptual model, we identify that the behaviour is expressed as a state machine in which transitions can change the state of the object. Invoking such a transition is the responsibility of either an internal actor or an external actor. In case of the latter, tasks can be defined to describe the interaction with the user and system such that the transition can be invoked. The definition of a transition, or more precisely, the specification of the arguments and the corresponding constraints can be used to generate tasks that enable the user to specify values for each argument.

7.1.1 Tasks

Arguments can be mapped to interaction classes. The type of argument determines which type of interaction is expected. As we defined our tasks as an input/output interface, mapping an argument to a specified interaction class is straightforward using the following heuristics. For each interaction class, a task is defined.

- **Query and Select** When the argument specifies a reference to an external domain object, the user should be able to select that object. In the task model, we map such an argument to a system query and a user selection class. Whereas the user selection class requires a list as

input, the retrieval of such a list is a task for the system.

- **Generate or Manipulate** When the argument is a data type that does not require the user to select an option, this argument will be mapped to a generate or manipulate interaction class.

7.1.2 Data Flow

Generating the data flow is only possible if the purpose of these data is clear. In case a task model is generated to execute a transition, each data object is mapped to the corresponding argument. In that case the output of the tasks that produce this data should be connected to the task that invokes that command. However, the output of a task may be used in a different task to enable users to verify if the data is correct. In that case, these data should also be connected to this specific review task. To be able to generate this data flow, the transformation generates a dedicated review task and connects the data flow to this task. We discuss patterns to configure such a transformation in Section 7.2.1.

7.2 Boilerplate approach

We discussed how arguments of a transition can be mapped to tasks in the task model. However, to generate a task model of an transition, we need to define patterns. The obvious approach would be to generate a single task for every argument that allows the user to specify a value for each. However, we identified common patterns that help the user to finish a task. We call this approach the boilerplate approach since it defines the basic sequence of tasks.

7.2.1 Workflow patterns/heuristics

Insert/Review/Confirm

In the initial state, the user is not able to perceive the state of the object other than perceiving that the object is not created. In that case, the user has to trigger the initial transition to create the object. For the initial transition, we can map the arguments to either data generation tasks and selection tasks. A review task can be generated to enable the user to review the values of the generated data and a confirm task can be generated to confirm the values and to trigger the invocation of the transition.

Select/Update/Confirm

For non-initial transitions, the user has to select the object that the user wants to view or manipulate. Depending on the state of the object, the user is able to select which task he wants to perform. Since these objects are created and contain information about the state of the object, this information can be used to help the user. For example, if the task should enable the user to update current attributes of an object, we can define a task with a *manipulation* interaction class with as input the current value of the attributes and as output the updated values. In that case, the user interface presents this information to the user such that the user can either manipulate the data or decide not to change the values.

7.3 Task model modularity

The boilerplate approach enables the generation of dedicated task models for transitions in a domain object. Since a task model for a single transition does not comprise the complete definition of tasks to be supported by an application, we defined an modular approach to compose a complete description of a task model for an application. In our approach we compose a complete task model by composing the task models we generated from the boilerplate approach.

7.3.1 Proxy Model

This modular approach defines a task model composed of proxy models. A proxy model contains a reference to an external task model. Whereas a *TaskDModel* is defined as a subtype of *Task*, each task model is also defined as an input/output interface. If a proxy task model is self-contained, we can use that model in the composition of a parent task model. A self-contained task model defines all required information to complete the task model either as system and user interaction or as input constraints and does not trigger tasks outside of the task model. Whereas the proxy model defines an input and output interface, the adopting task model (the parent) should comply with the input constraints of the proxy model. When the parent complies with the input constraints, and the proxy is self-contained, the output of the proxy task model can be used in the parent task model.

The advantage of this approach is threefold. First, when a transition changes, we can anticipate on that change by either generating a new version of that task model or changing the specific task model by hand. Secondly, proxy models can be nested. An example of a use-case of nesting task models is when in the middle of a task, the user has to select an object that does not exist yet. We could just abort the task and require the user to navigate to the creation task, but referring to the proxy task model that allows the

user to create such an object without forcing the user to abort his possibly long running process. In this way, we can nest proxy models to improve the usability of the user interface. Thirdly, we can define task models for specific users. By defining the capabilities in terms of transitions, we can define a task model for a specific user. This complies with the *Correctness by Construction* approach by avoiding repetition as we defined in Section 3.3.

7.3.2 *Application Configuration Model*

In the conceptual model, users are described as an entity with a set of capabilities. A capability refers to a specific transition of a domain object. This user model can be used to compose a task model by using the derived task models of each transition in the set of capabilities. A task model is composed by defining a parent task model and an application configuration definition. For each transition in the set of capabilities, proxy task models are derived, which will be stored in separate models. The parent model defines the main structure of the application. The application configuration specifies the set of users, the parent model and a node in the parent model to insert the proxy task models. For each user, a task model is composed and forms the input for the transformation chain. The metamodel of the application configuration model is depicted in Figure 7.1 which includes the user model as well.

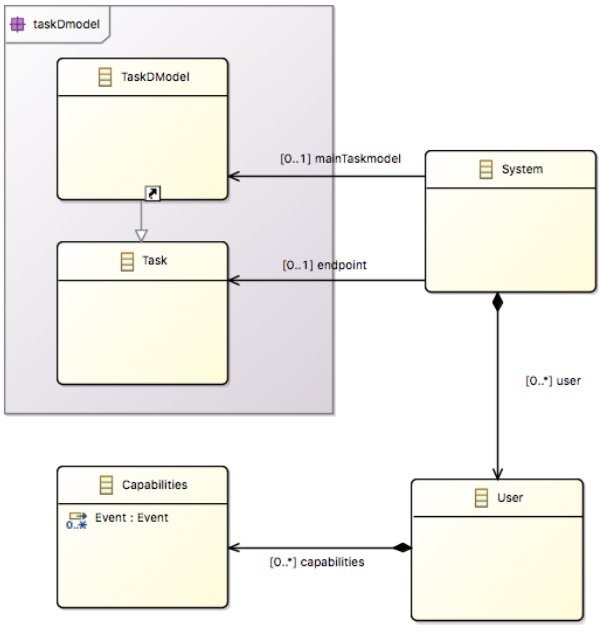


Figure 7.1: Metamodel of the application configuration definition model

7.3.3 *Transformation Definition*

To compose a task model based on an application configuration model, we define two transformations. The structure of the first transformation is depicted in Figure 7.2. The input of this transformation is an instance of a application configuration model and an instance of the domain objects. As we have discussed in Section 7.1, we can generate task models for each transition based on the specification of a domain object. This transformation uses the set of capabilities of each user to generate separate task models.

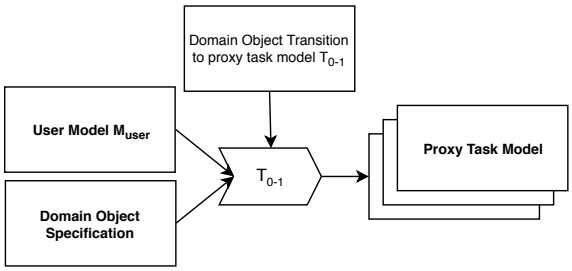


Figure 7.2: Input and output models of the first part of the transformation T_0

The second transformation uses the composition of the parent task model and the proxy models to create a task model for each user in the application configuration model.

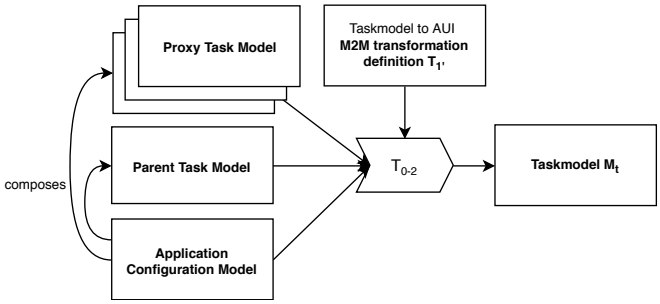


Figure 7.3: Input and output models of the second part of the transformation T_0

7.4 Limitations & Constraints

- The discussed approaches to derive a task model from a behavioural domain model are straightforward; a task is defined for each argument. Since this enables the user to enter a value for each argument, this can become tedious and is not user-friendly; the user has to go through every single step. More advanced transformations can detects groups of arguments and create logically grouped tasks.

- The current transformation defines a task for each argument. However, not every argument maybe mandatory for the transition to be triggered. In that case, either the arguments should be annotated as mandatory, or the user should configure the transformation to specify which arguments should be converted to a task.
- Since task models can be reused by different users with different user roles, the definition of these task models should take this into account. If users have different roles, executing a task may involve different queries and different steps to achieve the same goal.

7.5 Conclusion

In this section two approaches have been discussed to leverage behavioural domain models to integrate the behaviour of the domain models in the user interface. A transformation is defined to generate task models from behavioural domain models. This transformation generates tasks and interaction classes for each argument of the transition. Whereas the transformation is straightforward, the order of the tasks to be executed may not be considered as user-friendly. For this reason, more advanced transformations have to be defined. To anticipate on changes in the domain model we have discussed an approach that generates separate models dedicated to specific transitions. When a transition changes, only that specific transition has to be altered, and not the complete task model. In addition, an application with different users can be modelled and used as input for the generation of different task models for different users for the same application.

Chapter 8

Enforcing Correctness

In this chapter we discuss methods to enforce the correctness of our transformation chain results. We discuss methods we have used and that can be used to further verify correctness.

8.1 Correctness

In software systems, formal verification is a method of proving that the system is correct with respect to a certain formal specification or property. This proof is carried out on an abstract mathematical model of the system. Depending on the nature of the system, and the target properties, different mathematical models can be used. Examples are Finite State Machines, Labelled Transition Systems, Petri Nets and Process Algebra.

To reason about correctness, Hoare's logic is used, as first defined in [66]. Hoare defined a formal system to reason about the correctness of programs. The central notion of Hoare's logic is Hoare's Triple as depicted in Listing 13: S is the program in the form of statements that implement the function it executes, P is the precondition and Q is the postcondition, both in predicate logic. Using this Triple, partial correctness can be proven. *Partial correctness* gives the guarantee of a valid output only if the program terminates and the preconditions are met. However, to prove *total correctness*, in addition to partial correctness, one has to proof that the program always terminates.

$$\{P\}S\{Q\}$$

Listing 13: Hoare Triple

We consider model transformations as programs that operate on instances of the input metamodel. In this sense, we can apply the notions of correctness of programs on model transformations. We consider the input and output models as correct if they conform to structure and the constraints as specified in the metamodel. Therefore, we consider the structure and constraints of the source model as preconditions of the model transformation and the structure and constraints of the target model as postconditions.

The challenge is to proof that if the preconditions are met, the postconditions are guaranteed.

Different approaches use the notion of a *transformation model* to use existing model checkers to asses the correctness of the transformation. Cabot et al. defines an algorithm to derive a transformation model that consist of the source and target metamodel, and invariants in OCL [67] that are automatically derived from the transformation definition [68]. A model satisfiability checker (UML2Alloy [69]) is used to transform the transformation model to the specification language Alloy [70] to verify partial correctness. They have shown that this method can be applied for QVT [71], TGG [72] and ATL.

8.2 Validation Properties

To be able to assert the correctness of the intermediate models, we defined properties of a valid input and output model. These properties are based on the specification, in our case, the task model. We strived to define a set of properties that if this set of properties holds for every intermediate model, the models are correct and can result in an operational user interface that behaves as specified in the task model and according to the manual intervention.

8.2.1 Task model

As the task model forms the input for the transformation chain, this task model forms the specification for the resulting user interface: the user interface should support the workflows that are specified in the task model. We defined a number of properties of a valid task model:

1. **Task reachability** Tasks should be reachable if the execution is required in a specific workflow.

2. **Sinkholes** Is there a task where there is no point of return? As each task is defined as an input interface with preconditions, there should be at least one task where the preconditions can be met.

3. **Data availability** Can the input criteria of every task be met? As each task is defined as a function with preconditions, the input data must be produced before the user can start the execution of the task.

4. **System interaction** Is the system able to produce the output from the input? As we are not able to check the implementation of the system tasks, we can not verify if the implementation is correct. We can however check if the queries and commands are well-formed with respect to the specification of the domain models.

8.2.2 *User Interface Models*

The user interface models are derived from the task model. We require that these models preserve the correctness properties, and, in addition, we want to check if the semantics are preserved.

1. **Workflows preservation** Can the workflows be executed in each intermediate model? To verify this property, an approach would be to derive the workflows of the task model and check if intermediate models implements the workflows.

2. **Undesirable loops** Does the user interface contain behaviour that results in a loop? Since behaviour of the user interface is modelled with EventListeners, this behaviour can result in a loop.

8.3 OCL Constraints

A method to constrain valid models is to define constraints with the Object Constraint Language (OCL). These constraints are defined on the metamodel level and constraints the values of the features of an object. It is therefore limited to expressing local constraints. We show how these constraints can be used to enforce the correctness properties.

8.3.1 *Task model*

The invariant in Listing 14 constraints *Relationship* objects to define a relationship for tasks that are defined at the same level. To avoid dangling tasks, we define an invariant in Listing 15 to makes sure that each task is at least related to another task. As we do not distinguish abstraction tasks and interaction tasks in the metamodel, as we defined in Section 4.4, we verify this property by adding the invariant as listed in 16. By defining these constraints we prevent that tasks are unreachable due to dangling tasks (Property 1 and 2).

```
1  invariant relationship_between_tasks_with_same_parent:
2              self.relationships->forAll(r |
3                  (r.left_sibling.oclIsKindOf(AbstractTask) implies
4                  children->exists(t | t = r.left_sibling)) and
5                  (r.right_sibling.oclIsKindOf(AbstractTask) implies
6                  children->exists(t | t = r.right_sibling))
7              );
```

Listing 14: Invariant to constrain relationships to only occur on the same level

```
invariant all_tasks_must_have_a_relationship:
            children->forAll(t |
                relationships->exists(r |
                    r.left_sibling = t or r.right_sibling = t
                )
            );
```

Listing 15: Invariant to constrain subtasks

```
invariant only_interaction_on_leaf:
            if self.subtasks->isEmpty() then
                self.interaction->notEmpty()
            else
                self.interaction->isEmpty()
            endif;
```

Listing 16: Invariant to constrain interaction classes only on leave tasks

Data constraints

As each task is expressed as an object with an input and output interface, we want to enforce that the input conditions can be met. For each task we can enforce that the specified input and output variables meet the requirements to execute the interaction. To be able to execute the interaction, the input and output variables should be specified, the type of the variable should comply with the type of interaction and the input should be originated from a task that either retrieved or generated that data.

```
invariant correct_input_manipulation:
    if (self.type = '#MANIPULATION') then self.input->notEmpty() else
 ↪  true endif;
invariant correct_output_manipulation:
    if (self.type = '#MANIPULATION') then self.output->notEmpty() else
 ↪  true endif;
invariant correct_input_generation:
    if (self.type = '#GENERATING') then self.input->isEmpty() else true
 ↪  endif;
invariant correct_output_generation:
    if (self.type = '#GENERATING') then self.output->notEmpty() else true
 ↪  endif;
```

Listing 17: Invariant to constrain the input and output conditions for interaction classes

These invariants enforce that the referenced data objects correspond to the specified arguments to invoke the transition. However, it does not enforce that in the workflows the user is able to specify the values of these input objects (Property 3 and 4). Neither does it enforce that if the specified conditions can be met. To enforce this property we have to check the output of the preceding tasks.

```
invariant all_arguments_to_trigger_transition:
    let args : Integer =
        self.input->collect(i |
            self.event.arguments->exists(a | a.key.name = i)
        )->size()
    in not (args < event.arguments->size());
```

Listing 18: Invariant to constrain the input condition for commands to check if all arguments are provided for the specific transition

8.3.2 User Interface Models

The user interface models are less constrained compared to the task model. Whereas the task model is structured as a tree, the user interface models do not enforce such a structure. Therefore, the user interface models are not constrained on the structure. Only constraints on the data bindings of the interaction classes are defined.

Data binding constraints

The constraints on the data bindings are equivalent to the constraints defined on the task model.

```
invariant event_of_domainObject:
  type = SystemInteractionType::COMMAND implies
  ↪  self.domainObject.events->includes(self.event);

invariant list_with_list_input:
  self.input->notEmpty() and self.input.type = maverick::DataType::List;

class InputElement extends Element
    {
        invariant always_output:
            self.output->notEmpty();
        invariant type_check:
            (self.output->notEmpty() and self.input->notEmpty()) implies
  ↪  (self.input.type = self.output.type);
    }
```

Listing 19: Data binding constraints in the abstract and concrete user interface models

The defined constraints in OCL are not sufficient to assert if the models are valid with respect to the correctness properties. Other validation methods are needed, with a larger scope compared to OCL, to enforce if a combination of objects is valid. For example, with the OCL constraints we can not enforce that if the input data is generated in a workflow before reaching that particular task, nor can we enforce that if the user interface models support the same workflows as in the task model. For that purpose we can

convert the task model to a petri-net.

8.4 Validation Tools

Whereas the task model is based on the Task Oriented Object Design approach, as defined by [73], we can use a stronger validation method to check the correctness properties of the task model. For that purpose, the task model is converted to an Operational Petri-Net. Using this formalism, simulation can be carried out to validate and verify the dynamic characteristics of the task model. Existing tools like PetShop [74], Great SPN [75], and ReNew [76] can be used for this purpose. To translate the defined notation for the task model to the notation of TOOD, we have to investigate if we can add the notions of *synchronisation*, *prioritisation* and *coherence*.

8.5 Limitation & constraints

- OCL constraints are limited in verifying the correctness properties. For this purpose, other validation techniques are required to enforce the correctness of the models. As the task model is based on an existing technique which can be converted to an Operational Petri-net, this should be possible.
- The user interface models are not based on a formalism, therefore, we can not use available formal methods to verify the correctness of this model.
- The current defined constraints do not enforce that if the state changes, defined by the event listeners in the user interface models, are valid. As we can define the behaviour of a construct element in a state machine, it should be possible to statically analyse if the current state of the construct element allows the state change in the event listener.

8.6 Conclusion

The defined constraints are not sufficient to fully guarantee the correctness of the models. It only gives an indication about the wellformedness of the models. To validate the models, the Task Oriented Object Design approach enables the derivation of an Operational Petri-Net (OPN). Further research is required if this method is able to guarantee the correctness of the task model.

Chapter 9

Validation

In this section we discuss how different properties of the transformation chain, presented in chapters 3 till 8, have been validated. We discuss the goal, the method and our validation experiments.

9.1 Goal

The goal of the validation in this work consists of two parts: validate if business people with a technical background are able to specify the input of the transformation chain (Requirement 2) and validate that if a correct and operational user interface can be generated (Requirement 1 and 4) that can retrieve information and invoke commands of the system as described in Section 3.2 (Requirement 3). Because implementing a functional and correct transformation chain involves most of the effort in this research, we discuss this in more detail.

For the transformation chain we want to demonstrate if the models are able to express the concepts at the appropriate abstraction levels (Section 4 and 5). Furthermore, we want to validate if models of adjacent abstraction levels can be automatically transformed using the model transformations (Section 6). At last, we want to validate if we can generate a task model from a behavioural domain model (Section 7) and we want to validate to what extent the model transformations produces valid models. Concretely, we want to provide answers to the following questions, derived from the requirements (Section 1.3):

1. Can the models *express* the concepts at the appropriate abstraction levels?

2. Can we *transform* concepts from adjacent abstraction levels?

3. Are we able to *generate* a task model from a behavioural domain model?

4. Can we generate a task model that is *composed* by other task models?

As we defined a comprehensive set of features of an end-to-end process, from specification to an implementation in a specific implementation tech-

nology, we implemented a limited set of features such that we are able to demonstrate the basic functionality of the transformation chain from start to finish.

9.2 Method

To validate the transformation chain, we demonstrated its properties by executing it on real-world examples of behavioural domain models in three experiments. These models encapsulate the behaviour of financial services at ING, a large financial institution.

9.2.1 Implementation

For the implementation of the transformation chain we first parsed the textual specification of the behavioural domain models. For this process, we defined a metamodel that captures the features of the specification that are essential for the transformation chain. To parse and inject the specification into a model, a grammar is defined that matches the concrete syntax of the specification language. The injected models are used to derive a task model directly from a specification. Secondly, the metamodels of the task model, and UI models are defined using Ecore, which is implemented in the Eclipse Modelling Framework (EMF) to support the development of modelling tools. Model transformations are defined in the Atlas Transformation Language (ATL) and executed with the EMF framework.

The Final UI is implemented in a Javascript framework. Using such a framework enables fast prototyping as it takes care of retrieving and setting data with data bindings. The implementation of the Final UI is presented in Appendix F.

9.2.2 Experiments

A combination of a top-down and a bottom-up approach are used in the conducted experiments. We used a top-down approach to generate an implementation from individual behavioural domain models to check if an implementation can be generated for every domain model that is provided. Because we implemented a rather straightforward transformation from behavioural domain model to task model, more advanced task models are produced by hand that uses a broader set of features of the task model. These more complex examples are produced using a bottom-up approach of reverse engineering real-world examples from the financial domain. In cases where the defined transformations are not able to automatically generate certain concepts, we show that the models do account for the discussed features by editing these models by hand. Although the

transformations are not able to generate the necessary concepts, we can show that the models do support these features. We conducted the following experiments:

Experiment 1. Exhaustive specification test
Execute the entire transformation chain on each individual transition of the domain models and use this as input for the transformation chain to generate a Final UI that supports the steps to invoke the transitions.

Experiment 2. Simple example: Money transfer
Creating a simple example of a more complex case from a typical business process, as input for the transformation chain to generate a Final UI.

Experiment 3. Real-world case: BuyerFunderAgreement
Reverse engineering a mock-up from a real-world use-case and specification of behavioural domain model to generate the functionality to implement the mock-up.

To evaluate the correctness of the transformation chain, we defined a number of correctness indicators. Transformations are evaluated by the number of warnings during the transformation, the amount of violated constraints in the target model, the amount of manual operations and if the transformation terminates. Since the last indicator is obvious, we only discuss transformation warnings and the violated constraints.

Transformation warnings

The number of warnings indicate if some elements are not transformed as the metamodel of the target model expects. Throughout the transformation chain different warnings can be reported. For example, a *cardinality mismatch* occurs when the transformation adds a collection of elements to a single-value feature. This error might introduce nondeterministic behaviour as the transformation engine decides which element is assigned. Therefore, elements might be skipped during the transformation process and that is not desirable. An *inter-model containment* warning can occur when elements of the source model are not transformed during the process, but referenced to in the target model. This could lead to undesired outcomes and can be propagated to other levels of the transformation chain. An *overlapping feature* warning could occur when the source and target model define a feature with the same name. This is problematic when features have the same name, and should contain the same properties, but are not consistent.

Violated constraints

Model constraints are defined to evaluate if the models are well-formed. While this is a property of a model, it is a correctness indicator of a transformation as the transformation process should output a valid model. While the well-formedness of a model is an indication of a valid model, we use it to evaluate the transformation definition.

Amount of manual operations

While the defined transformations can automatically generate elements, some models need to be adapted by hand. Although the goal is to fully automate the transformation chain and to configure the model transformations to adapt the user interface, the current defined transformation requires some minor alterations such that the generated implementation integrates the required properties. The amount of manual operations gives an indication to what extent the models and the transformation support the automatic generation of the required features.

9.3 Exhaustive specification experiment

For this experiment we have collected a set of 16 specifications of behavioural domain models. These specifications are used for a specific application in a real-world case in the financial world. For this experiment, we have automated the transformation chain by defining *ANT tasks* for each model transformation in the chain, and saved the intermediate models for further inspection. Because the last transformation to the final user interface is not integrated in the ANT workflow, only a sample of the generated concrete user interface models is transformed to a final user interface. This final user interface is evaluated visually in the web browser and tested by executing the task models. For each behavioural model, we have collected the correctness indicators.

In this experiment we observed the following:

Observation 1. All transformations terminate and resulted in a target model.

Observation 2. The user interface models do not violate any major constraints. Only a minor violation has been observed of a transformation that created a redundant *Relationship* object.

Observation 3. Models in the transformation chain contain overlapping enumerators but the literals in these enumerators are consistent.

Observation 4. The transformation AUItoGUI elimates eventlisteners that are triggered by the event that it raises.

9.3.1 *Minor violation*

The minor violation, from Observation 2, is the result of the first process of generating a task model from a domain model. As this is not a desired outcome, the transformation should be altered such that it detects the last task and thus it does not create a redundant *Relationship* object.

9.3.2 *Redundant EventListeners*

In the process of transforming the task model to the abstract user interface models, redundant eventlisteners are created (Observation 4). These Eventlisteners listen to the same event that they emit. This is a problem as it might result in a loop. The cause of these problems lie at the transformation definition which is defined in a single step. Therefore, as a result, temporary properties are assigned, like in this case, events. A solution would be to create a two step transformation such that these objects are not created or that these objects will be removed.

For now, this problem is obfuscated as the next transformation detects and eliminates these objects. In addition, the final user interface implements functions that detects and prevent these kinds of loops. Nevertheless, the abstract user interface models are not valid when these objects exist, and as we have defined in Section 3.3, this problem needs to be solved at the root.

9.3.3 *Overlapping enumerators*

Multiple intermediate metamodels contain enumerators with the same name which cause the *Overlapping enumerators*. For now, we were able to check the consistency among these enumerators, we conclude that this did not impacted the results. However, when models needs alterations, the consistency across the metamodels is difficult to maintain. A maintainable solution, which mitigates these consistency problems, would be to extract common enumerators in separate models. The development environment that we used had problems with cross-references in models, and therefore this solution has not been implemented.

9.4 Simple example: Money transfer

The money transfer example, which formed the running example in this research, is used as an example of a single purpose task model. Its solely purpose is to describe workflows for a single user to enter information related to transferring money from a bank account of the user to another

bank account. Compared to a task model directly derived from the specification, this example is more advanced as it incorporates more than a single workflow. We have used the real-world specification of an *InhousePayment*, which specifies the behaviour of a payment process. While this specification is developed as such that it should be implemented in a real-world application, the specification is not considered as final. The task model of the toy example is presented in Appendix E.

In this experiment we observed the following:

Observation 1. All transformations terminate and resulted in a target model.

Observation 2. A significant number of warnings report overlapping features in each transformation. This problem is related to the problems with cross-references as described in Section 9.3.3.

Observation 3. For unknown reasons, the mapping model contains an unresolved proxy to the abstract user interface model. After resolving this proxy by hand, the problem was solved. This problem is related to the problems with cross-references as described in Section 9.3.3.

Observation 4. The *EventListener* object, contained by the container that offers the user to choose between two options, does not define the fact that when a choice has been made, that specific container should hide. Manual intervention is required to solve this problem.

Observation 5. The task model violates the constraint that either an interaction class should be assigned to a task or the task must be decomposed with an interaction class. For a particular container, this constraint has been violated and therefore the transformation should be improved.

The root of the problem of Observation 2 and Observation 3 is the problem with cross-references as described in Section 9.3.3.

9.4.1 Choice Container

In our approach the structure of the task model can be defined using temporal constructors. Based on this structural model, an event-driven task model can be derived. Whereas this process is straightforward, and results in a semantically valid model, the defined constraints do not consider this as a valid model. This is observed in the transformation from a *Relationship* object with the temporal constructor *Choice* to an event-driven structure. The root of the problem is that we have chosen for defining a separate task that defines the triggers to execute one of the two choices to implement the choice. This task is generated in the process of converting a structural model to an event-driven task model. While the rest of the tasks are converted, the task that allows the user to make a choice should be injected as

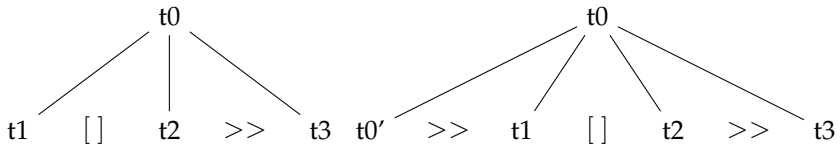
transformation	input (xmi)	output (xmi)	warnings
ExpressTemporalRelationships	InhousePayments.taskd	InhousePayments.taskd.refined	20
TaskDmodelToAUI	InhousePayments.taskd.refined	InhousePayments.aui InhousePayments.mapping	23
AUItoGUI	InhousePayments.aui InhousePayments.mapping	InhousePayments.cui	20

Table 9.1: The transformations involved in the *InhousePayments* experiment

model	type (metamodel)	auto/manual	violated constraints	notes
InhousePayments.taskd	Taskmodel	manual	0	
InhousePayments.taskd.refined	Taskmodel	auto	1	The container that enables the user to choose which task he wants to perform violates a constraint.
InhousePayments.aui	AbstractUI	auto	0	
InhousePayments.mapping	Mapping	auto	1	The automatically generated mapping model contains an unresolved proxy and is thus not valid.
InhousePayments.cui	ConcreteUI	auto	0	The choice container does not hide, when a choice has been made.

Table 9.2: The generated models involved in the *InhousePayments* experiment

an event-driven task. Considering that this transformation is defined in a single step, the logic that defines which tasks reacts on which event has to be defined specifically for that task. In addition, converted tasks have to be altered such that the injected task will be triggered. The complexity of injecting a task in a single step with a declarative transformation definition lies at root of the problem we have seen in Observation 4 and Observation 5. An approach that solves this problem is by separating the injection of the choice task and the process of generating event-driven tasks. The first step is depicted in Figure 9.1.

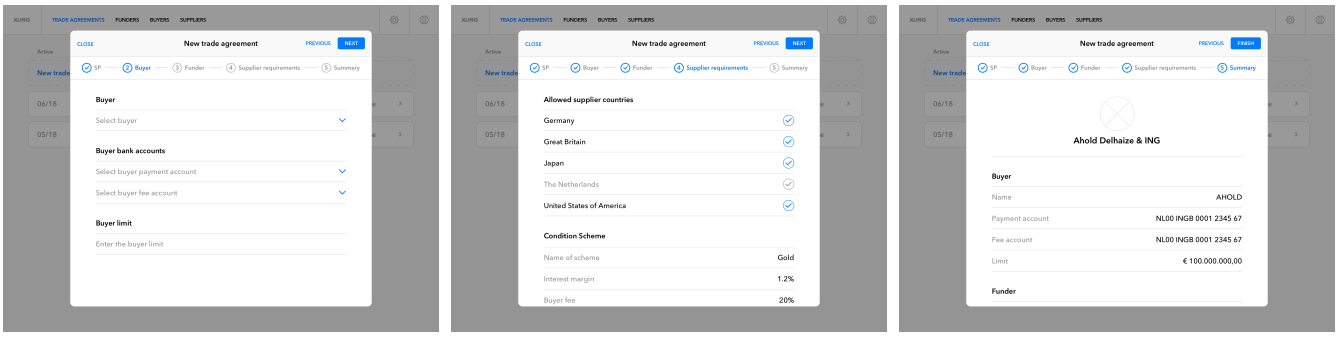


(a) The original task model tree (b) The generated tree

Figure 9.1: Injecting the choice task

9.5 Real-world case: BuyerFunderAgreement

To validate if the transformation chain can produce user interfaces for a more realistic application, a bottom-up approach is used to create a task model from a set of mock-ups. This set of mock-ups is designed independent of this research without the intention to reverse-engineer the behaviour in a task model. An excerpt from the mock-ups is depicted in Figure 9.2. The mock-ups comprise a process that enables the user to define the necessary tasks to create a *BuyerFunderAgreement*, which is basically an agreement between a buyer entity and a funder entity. These tasks involve the selection of the involved entities and defining the agreement conditions.



(a) (b) (c)

Figure 9.2: Mock-ups for the BuyerFunderAgreement application

For this example, a behaviour domain model, the *BuyerFunderAgreement* specification, is available. This model defines 8 events, triggering transitions that range from creating an object (create) to changing the state to *accepted*. From the domain specification, we derived tasks for each argument in the transition *create*. Whereas the mock-ups define the process of creating an agreement in 5 steps, for each group an abstraction task is defined that contains the smaller subtasks. These tasks are adopted in a task model that simulates a realistic application. An excerpt of this model is depicted in Figure 9.3. The three main tasks are: *Authenticating*, *Manage Agreements* and *Logout*.

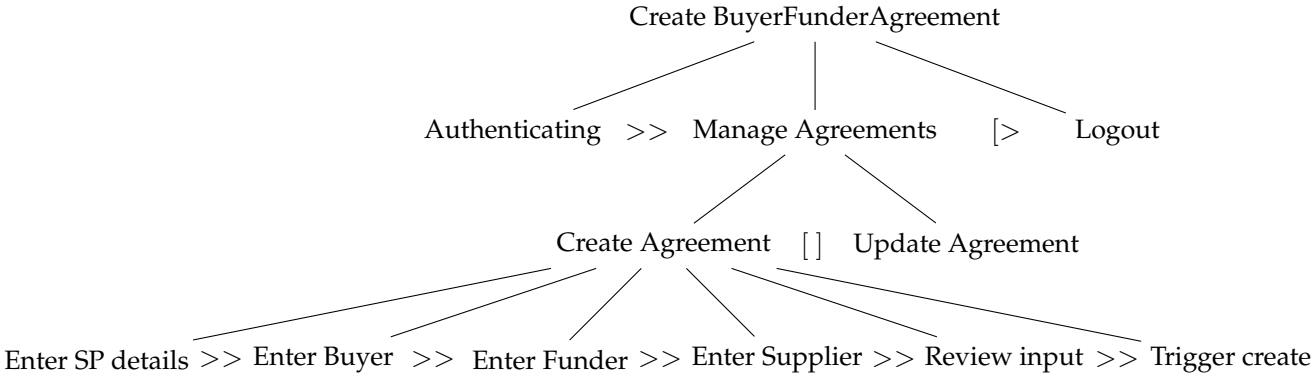


Figure 9.3: Excerpt from the *Create BuyerFunderAgreement* task model

In addition to the observations from the previous experiments, in this experiment we observed the following:

Observation 1. All transformations terminate and resulted in a target model.

Observation 2. Manual operations were necessary to include a reaction that enables the user to go to the previous task, as defined in the mock-ups.

Observation 3. Automatically generated titles and text for buttons are not suitable for a real-world application.

Observation 4. The state of construct elements changes during the interaction with the user and this state is not saved. The current implementation only saves the state of containers to toggle the visibility.

transformation	input (xmi)	output (xmi)	warnings
ExpressTemporalRelationships	Agreement.taskd	Agreement.taskd.refined	21
TaskDmodelToAUI	Agreement.taskd.refined	Agreement.aui Agreement.mapping	23
AUItoGUI	Agreement.aui Agreement.mapping	Agreement.cui	40

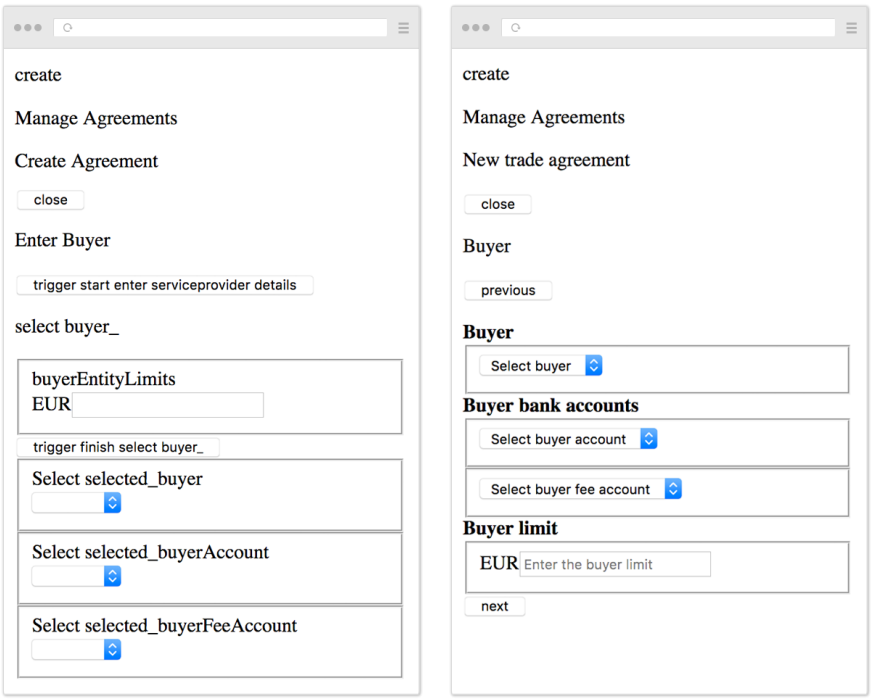
Table 9.3: The transformations involved in the *BuyerFunderAgreement* experiment

model	type (metamodel)	auto/manual	violated constraints	notes
Agreement.taskd	Taskmodel	manual	0	
Agreement.taskd.refined	Taskmodel	auto	0	The reactions that enable the user to go back to the previous tasks are added by hand
Agreement.aui	AbstractUI	auto	0	
Agreement.mapping	Mapping	auto	1	The automatically generated mapping model contains an unresolved proxy and is thus not valid.
Agreement.cui	ConcreteUI	auto	0	

Table 9.4: The generated models in the *BuyerFunderAgreement* experiment

9.5.1 *Automatic generated text elements*

As the implementation in a graphical user interface often requires text to, for example, label form elements or to indicate sections, text elements are generated. These text elements however, contain values that are generated throughout the process which are not necessarily suitable for presenting information in a graphical manner, as described in Observation 3. In Figure 9.4, we illustrated the difference between the generated implementation (9.4a) and the desired implementation (9.4b).



(a) (b)

Figure 9.4: Screenshot of the generated implementation and the desired implements (without style)

It is obvious that the generated text elements in Figure 9.4b are not suitable. To solve this, an approach would be to define a model that maps the relevant text elements to a more appropriate value. This would form the input for the transformation from an abstract to a concrete user interface model. Since for each modality the communication and interaction with the user is different, this process is different for each modality. For example, in voice assistant user interface the concept of sections and form elements do not exist. Therefore, these text elements are transformed either to statements, in the case of separate sections, or to questions to indicate input elements. In addition, this approach would make it possible to support different values for different languages. The approach does, however, involve manual intervention whereas our goal is to automate the process.

9.5.2 *Manual intervention to include back button*

In the workflow of creating a trade agreement, an option should be adopted that enables the user to return to the previous task. However, as the structural model specification does not support the definition of such an option, manual intervention is required to include this option in the dynamic specification. To avoid this additional step, a solution would be to annotate the task in the structural model that should include this option. The transformation should add a *Trigger* and an *EventListener* object that hides the current tasks and enables the previous task when the user triggers the trigger.

9.6 Usability of GLUI

Although we require business analysts with a technical background to be able to understand and to define the specification of a task model, the usability of the task modelling language should be validated. During the research, proposals for a concrete syntax have been discussed in an informal setting with a business analyst. This proposal can be found in Appendix G. Because the task model has been evolved during the research, and the concrete syntax has not, we have not been able to fully define a concrete syntax for the current state of the task model. Therefore, we have not been able to fully validate if business analysts are able to understand and use the concrete syntax to define a task model specification. Also, because the business analyst was involved in the research as a supervisor, his judgement might be biased. Nevertheless, the first signs are positive since the business analyst was positive towards defining the task model using the proposed concrete syntax. To fully validate the usability of the language, usability tests should be carried out with a larger population of business analysts with a technical background. Despite the fact that the proposal for a concrete syntax embodies a textual representation of the task modelling language, the notation of the CTT can be used for the structural model.

9.7 Conclusion

Because we have not implemented every feature of the transformation chain to be integrated in the Final UI, we can not validate every feature of the transformation chain. Based on the conducted experiments we are able to answer the questions stated at the beginning of this section. We can conclude that the models at the abstraction levels are able to express the concepts at each level. The construct elements defined in the abstract user interface can be transformed to refined elements in the concrete user interface model. Although the semantic gap between the concrete GUI user interface is not significant, an implementation can be generated from the

concrete user interface model. We have shown that the Final UI is able to interact with the user and the system, to implement the user interface logic in terms of event listeners and to implement the data binding required to store the values entered by the user and retrieved by the system. Whether the task model can be defined by business analyst should be further validated.

Chapter 10

Final Remarks

10.1 Conclusion

In this research we have investigated how to apply an MBUID approach that leverages characteristics of behavioural domain models to automatically generate an operational front-end application. For this purpose we have defined a transformation chain that defines three metamodels capturing concepts from the first three abstraction levels of the widely used Chameleon Reference Framework. Between conjunctive abstraction levels, model transformations successfully transform concepts from source to target model. A model-to-text transformation has been defined to generate an operational user interface that implements the functionality defined in the task model and the user interface models.

Customisation of the model transformation has been achieved by defining a mapping model that maps abstract construct elements to concrete elements. This enables human intervention and stores design choices in a separate model such that the transformation results in the same customised model and design choices are not lost.

To generate an appropriate user interface, we have discussed how the principles of Correctness by Constructions can be integrated. Based on these principles we have focused on a rigorous task model specification as well as correctness preserving transformations. Whereas the latter is based on constrains that determine a valid model, the enforcements of these constraints determine the correctness of this method. More research is required to verify the correctness of the model transformation.

We are able to leverage the behaviour of a behavioural domain model by deriving a task model from individual transitions. A transformation is defined to show that by using heuristics a task model can be derived. While for each transition a task model is derived, it can not be considered as complete description of a task model for an application. For that purpose, we have defined an approach that composes a complete task model based on a model of an application. Both approaches have been successfully used and resulted in implementing the behaviour in the user interface.

To validate the transformation chain, real-world behavioural domain models are used to generate an operational user interface that implements the behaviour of these models. An exhaustive test has been carried out to successfully generate a task model of each domain model specification. More advanced task models have been defined by hand to test more advanced features of the transformation chain. This did not lead to significant problems and resulted in an operational user interface.

10.2 Future work

Concrete Syntax To validate if business analysts with a technical background are able to understand the concepts and the relations of the task model, a concrete syntax should be defined, in both a graphical or a textual form. Whereas tools are available to define a Domain Specific Language based on metamodels, the derivation of a DSL is straightforward. However, whereas the learning curve of a DSL is a hurdle to overcome, a graphical notation would be more accessible. An hybrid approach would be another option to investigate. More research is required to determine which approach is suitable for this task model.

Verification of transformation definitions Although we have validated the transformation definitions based on the violated constrains of the source and target models, the correctness of the model transformations can not be guaranteed. For this reason, more research has to be conducted to find out if the mentioned method can prove if the transformation definition is correct. For this purpose, other model transformation languages, based on graph transformations, should be considered as more suitable methods exist that are able to prove this property.

Defining dedicated models for modalities In this research the abstraction levels of the Cameleon Reference Framework provided the guidelines for the concepts at each intermediate model in the transformation chain. Although, these models have been altered such that interaction can be defined more explicit, and the interaction with the system can be specified, further research is required to validate if concrete models for different modalities can be generated from the abstract user interface model.

Deriving a Petri-net from task model The dynamic task model is based on a method that uses an Operational Petri-Net to verify the correctness of the task model. However, although we have defined a simplified version of this task modelling method, further research is required how our version can be converted to a OPN. Furthermore, more research is required to validate if this method is useful to check the correctness properties we defined for the task model.

To obtain the metamodels and the transformation definitions, send a message to *peter1wessels@gmail.com*.

Appendices

Appendix A

Specification of Bankaccount and Transaction

```
specification BankAccount {
     fields {
         accountNumber: IBAN
        balance: Money
     }

     events {
         openAccount[minimalDeposit = EUR 50.00]
         withdraw[]
         deposit[]
         block[]
         unblock[]
         interest[maxInterest = 5%]
         close[]
     }

     invariants { mustBePositive }

     lifeCycle {
         initial init -> opened: openAccount

        opened -> opened: withdraw, deposit, interest
         -> blocked: block
         -> closed: close
         blocked -> opened: unblock

         final closed
      }
}
```

Listing 20: Specification of BankAccount domain model

Appendix B

Obtain Execution Traces Algorithm

The data structure of the task model allows different workflows defined in a single tree. We call the set of workflows supported by the task model: execution traces. These execution traces define the order of tasks to terminate the root node. That is, the user follows a trace such that the root node is considered as executed. The first step to obtain the execution trace from a task model tree is to convert the ConcurTaskTree to a concurrent decision tree as defined in Definition 1. From the concurrent decision tree we can follow the *InOrderTrace* algorithm as defined in Listing 23 to obtain the execution traces.

Definition 1. *We define a concurrent decision tree as a tree defined recursively:*

- *An unbounded set of symbols T are called tasks and denote a proposition that is either true when a task is executed or false if not. Items in the set T are denoted by a lower case t followed by any number in the set of Natural Numbers N.* $T = \{t1, t2, t3...\}$
- *A task is a node without children, a leaf, labeled by a symbol of set T.*
- *A decision is a node labeled by the symbol of the temporal constructors with two children both of which are either a task or a decision.*

$$t0 \iff t1 \vee t2 \vee t3$$
$$t0 \iff t1 \vee (t4 \wedge t5) \vee t3$$

Listing 21: Formula corresponding to Figure B.1a

To convert Figure B.1a to a concurrent decision tree, we follow the procedure in Listing 24. The procedure consist of 2 stages. In the first stage, it defines the temporal constructors contained by the root node as nodes in the converted tree, as in Figure B.1b. In the second stage, this process is repeated for each leaf of the result of stage 1 and results in Figure B.1c.

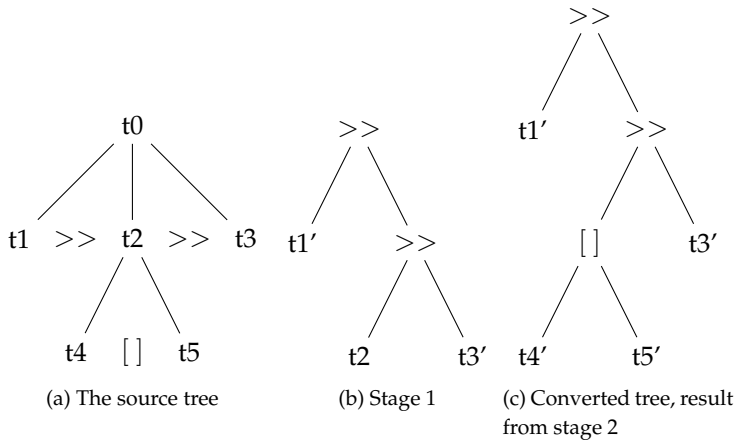


Figure B.1: Converting a ConcurTaskTrees to a concurrent decision tree

As we can see in Figure B.1, the first stage is to create a node for the temporal constructor of the first task in the list, in this case $t1$ is related to $t2$ with the *SequenceEnabling* constructor ($>>$). Then, we define separate branches for both tasks the temporal constructor relates, so in this case $t1$ and $t2$. For the right-branch, in this case $t2$, we repeat the process with the next task in the list, $t3$, and substitute $t2$ with the result of this process. This results in the branch with *SequenceEnabling* constructor ($>>$) as the label of the node and $t2$ and $t3$ as branches. This process is repeated until no items are left in the list. The result is illustrated in Figure B.1b. For each leaf in the tree of stage 2, $t1, t2$ and $t3$, we repeat this process if the leaf, in the current stage of the conversion, is a subtree in the original tree. In this example, $t2$ defines a subtree with two branches, $t4$ and $t5$. Therefore, repeat the process of Stage1 for the subtree of $t2$ and substitute the placeholder with the result of this process.

Using this conversion we can use an InOrder algorithm, as described in Listing 23, to obtain the execution trees. This algorithm uses the procedures defined for each temporal constructor, as defined in Table B.1.

1. t1 $\rightarrow$ t4 $\rightarrow$ t3
2. t1 $\rightarrow$ t5 $\rightarrow$ t3

Listing 22: Resulting set of traces

So, do the conversion and the defined procedure (Listing 23) enables us to obtain the traces? To answer this question, we have to verify if the resulted set of traces are valid. A property of a valid trace is that a user can execute the trace. We call these traces, executable traces. An executable trace defines a sequence of interaction tasks that can be executed one after each other while considering the structural model. So each task in the trace should be an interaction task.
When the CTT is well-formed, this property is guaranteed as the conversion from CTT to CDT, in Figure B.1, requires that the subtree of abstraction

Name	Procedure
$>>$	Store both branches in the same trace in order of appearing.
$[\,]$	Copy the current trace and store *InOrderTrace(t1)* in the current trace and *InOrderTrace(t2)* in the new trace.
$\|\|\|$	Copy the current execution trace and store in the current trace *InOrderTrace(t1)* followed by *InOrderTrace(t2)*, in the new trace *InOrderTrace(t2)* followed by *InOrderTrace(t1)*.
$\|=\|$	Copy the current execution trace and store in the current trace *InOrderTrace(t1)* followed by *InOrderTrace(t2)*, in the new trace *InOrderTrace(t2)* followed by *InOrderTrace(t1)*.
$\|>$	Copy the current trace and store in the current trace *InOrderTrace(t1)* followed by *InOrderTrace(t2)*, in the new trace, *InOrderTrace(t1)*.
$[>$	Store both branches in the same execution trace in order of appearing.

Table B.1: Procedures for temporal constructors

```
InOrderTrace (T)
  if T is a leaf
    add the label to the trace
    returns

  let t1 and t2 be the left and right subtrees of T

  follow the corresponding procedure of the root label (temporal constructor) of T
```

Listing 23: Recursive procedure to obtain the execution traces

```
Conversion(T):
  if T is leaf:
    create a node with its value as label

  // Stage 1
  let C be the ordered list of temporal constructors of T and i its index
  let s1 be Stage1(C)

  // Stage 2
  let L be the set of leaves of s1
  foreach l in L:
    if l is a node in T:
      substitute l with Conversion(l)

Stage1(C):
  let C[0] be the first temporal constructor on the list
  create a node for C[0] with the value as its label

  let t1 and t2 be the subtrees the temporal constructor relates
  create a left-branch for t1 with the value as its label
  if C[1] exists:
    pop C[0] from list
    create a right-branch for Stage1(C)
  else:
    create a right-branch for t2 with the value as its label
```

Listing 24: Recursive procedure converts a CTT to a concurrent decision tree

tasks, in this case *t2*, are to be substituted recursively by its equivalent concurrent decision tree. The result is that the CDT tree does not consist of any abstraction tasks, but only of interaction tasks and temporal constructors. We can also verify if the root node can be considered as executed by evaluating the corresponding formula. In Table B.2, we evaluated the formula corresponding to Figure B.1a. We can verify that the resulting traces evaluate the corresponding formula to *true* and others to false.

t1	t4	t5	t3	$t1 \wedge (t4 \vee t5) \wedge t3$
F	F	F	F	F
T	F	F	F	F
T	T	F	F	F
T	T	T	F	F
T	T	T	T	F
T	T	F	T	T
T	F	T	T	T

Table B.2: Truth table to determine if the task model is finished

The resulting set of execution traces can be used to verify if the task model consists of the workflows the modeller wants to include in the interface. In addition, as the execution traces are a property of the user interface, we can use this property to validate intermediate models.

Appendix C

Metamodel of Taskmodel

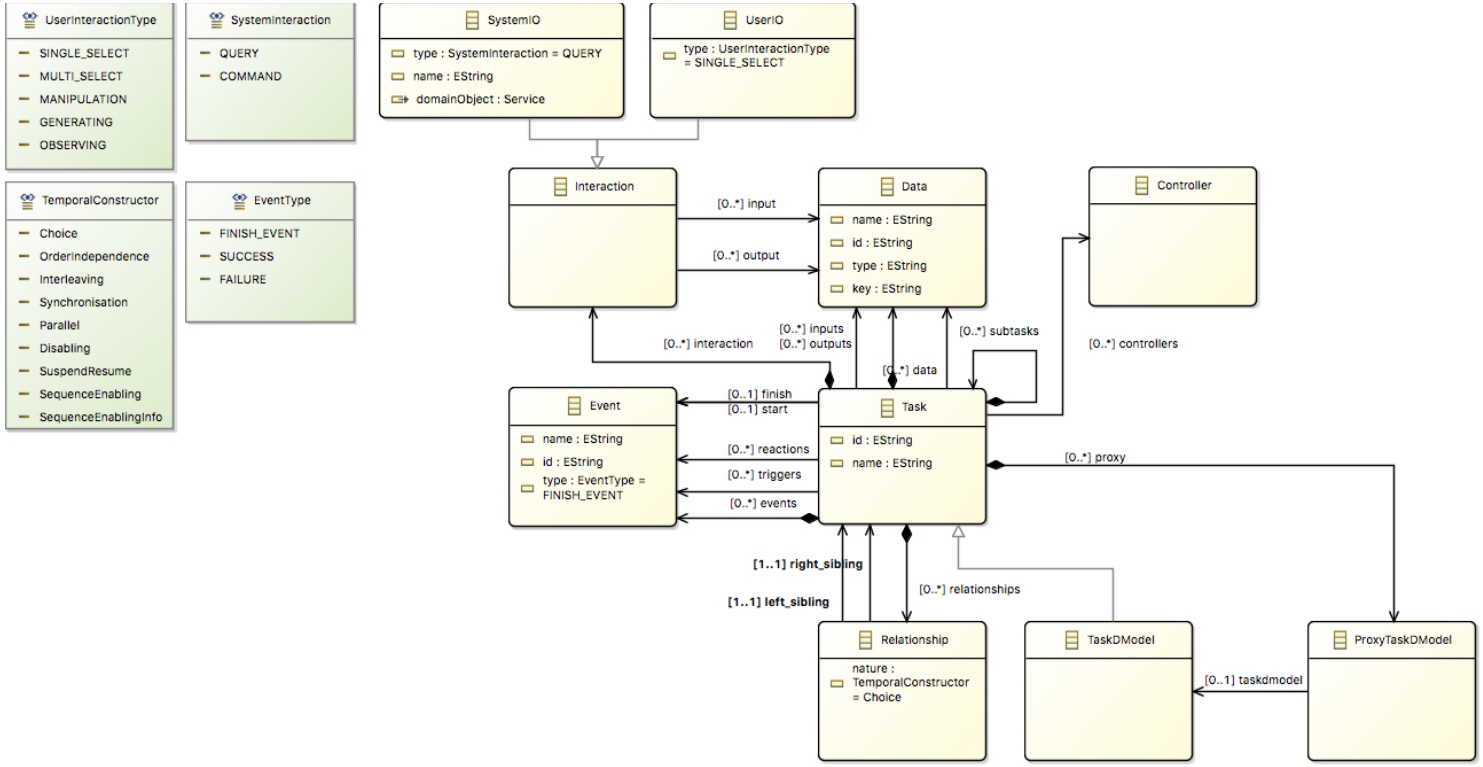


Figure C.1: Metamodel of the taskmodel

Appendix D

Concrete Graphical User Interface Metamodel

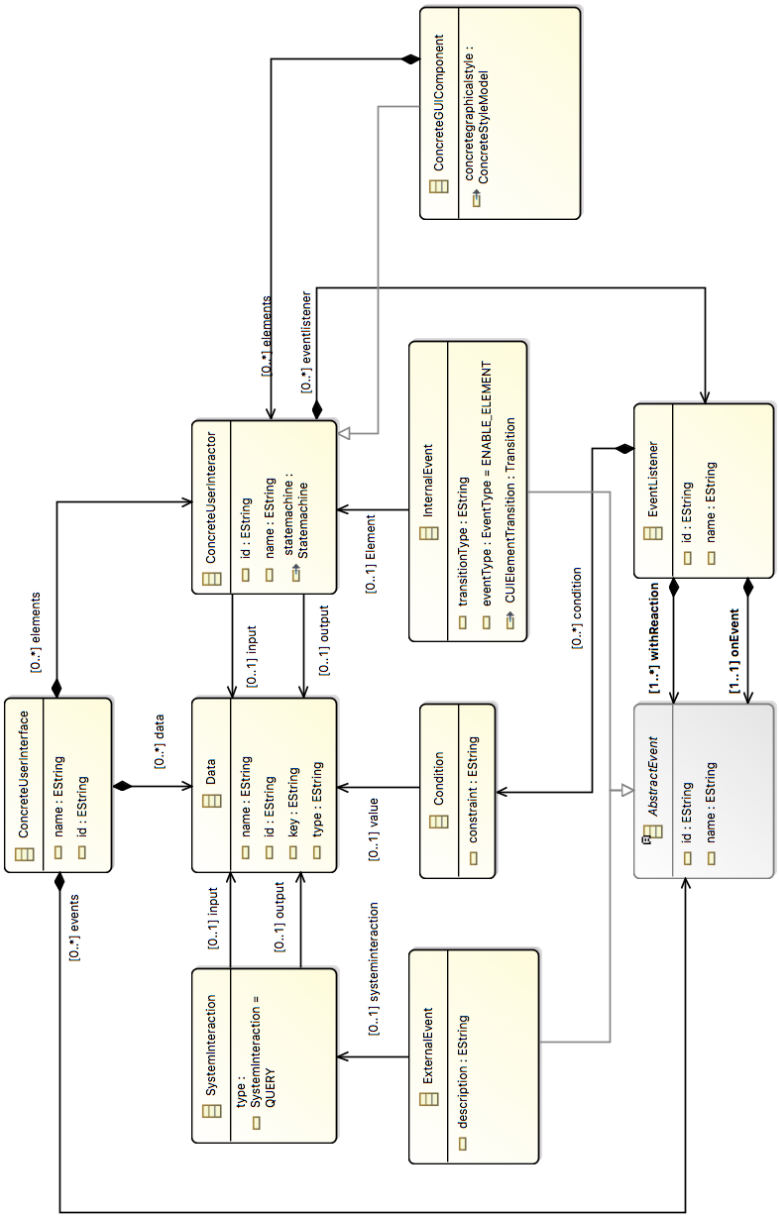


Figure D.1: Metamodel of the concrete graphical user interface models

Appendix E

Taskmodel of toy example: Money transfer

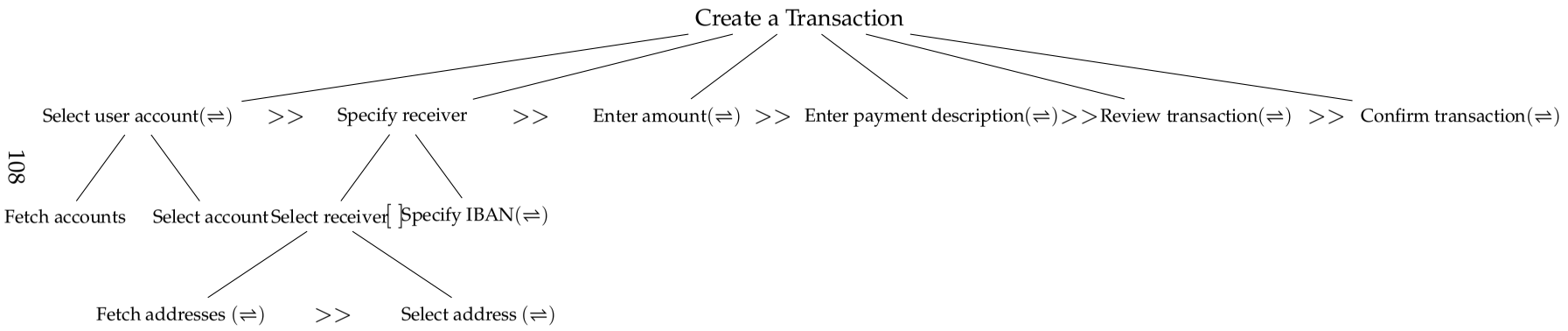


Figure E.1: Money Transfer task model

Appendix F

Implementation in VueJS

For every container, a component has been generated which contains the subcontainers. A vueJS store has been implemented that updates the state of the components. Interceptors are used to mock the calls and the callbacks of the system interaction.

```
<template>
  <div>
 <p> create </p>

  <select_debitAccount class="select_debitAccount" v-if="
    select_debitAccount_enabled" v-on:trigger="eventlistener($event)
    ">
  </select_debitAccount>

  <ChooseCredit class="ChooseCredit" v-if="ChooseCredit_enabled" v-
    on:trigger="eventlistener($event)">
  </ChooseCredit>

  <enter_amount class="enter_amount" v-if="enter_amount_enabled" v-
    on:trigger="eventlistener($event)">
  </enter_amount>

  <enter_paymentDescription class="enter_paymentDescription" v-if="
    enter_paymentDescription_enabled" v-on:trigger="eventlistener(
    $event)">
  </enter_paymentDescription>

  <Confirm_Transaction class="Confirm_Transaction" v-if="
    Confirm_Transaction_enabled" v-on:trigger="eventlistener($event)
    ">
  </Confirm_Transaction>

  </div>
</template>
<script>
import select_debitAccount from '@/components/select_debitAccount'
import ChooseCredit from '@/components/ChooseCredit'
import enter_amount from '@/components/enter_amount'
import enter_paymentDescription from '@/components/
    enter_paymentDescription'
import Confirm_Transaction from '@/components/Confirm_Transaction'

import eventlog from "@/helpers.js";
export default {
  name: 'create',
    computed: {

```

```
      select_debitAccount_enabled: function () {
      return (this.$store.getters.getState('ca6select_debitAccount2').state == 'ENABLE_ELEMENT')
    }
      ,
      ChooseCredit_enabled: function () {
      return (this.$store.getters.getState('ca3ChooseCredit28').state == 'ENABLE_ELEMENT')
    }
      ,
      enter_amount_enabled: function () {
      return (this.$store.getters.getState('ca12enter_amount14').state == 'ENABLE_ELEMENT')
    }
      ,
      enter_paymentDescription_enabled: function () {
      return (this.$store.getters.getState('ca13enter_paymentDescription17').state == 'ENABLE_ELEMENT')
    }
      ,
      Confirm_Transaction_enabled: function () {
      return (this.$store.getters.getState('ca5Confirm_Transaction32').state == 'ENABLE_ELEMENT')
    },
  },
    components: {
      select_debitAccount,
      ChooseCredit,
      enter_amount,
      enter_paymentDescription,
      Confirm_Transaction
    },

  data() {
    return {
        select_debitAccount : this.$store.getters.getState('ca6select_debitAccount2').state
      ,       ChooseCredit : this.$store.getters.getState('ca3ChooseCredit28').state
      ,       enter_amount : this.$store.getters.getState('ca12enter_amount14').state
      ,       enter_paymentDescription : this.$store.getters.getState('ca13enter_paymentDescription17').state
      ,       Confirm_Transaction : this.$store.getters.getState('ca5Confirm_Transaction32').state
    }
  },
  methods: {
    eventlistener(events) {
      for(var i = 0; i < events.length; i++) {
        this.$store.dispatch('update_state', events[i])
      }
    }
  }
}
</script>

<style>

  .select_debitAccount {
```

```
    background-color:red;
    display: block;
  }

  .ChooseCredit {
    background-color:red;
    display: block;
  }

  .enter_amount {
    background-color:red;
    display: block;
  }

  .enter_paymentDescription {
    background-color:red;
    display: block;
  }

  .Confirm_Transaction {
    background-color:red;
    display: block;
  }

</style>
```

Listing F.1: VueJS Component definition

```

import Vuex from 'vuex'
import Vue from 'vue'
import axios from 'axios'

Vue.use(Vuex, axios)

export default new Vuex.Store({
  state: {
    elements: [

      {
        element: 'ca1create26',
        state: 'DISABLE_ELEMENT'
      }
      ,
      {
        element: 'ca6select_debitAccount2',
        state: 'DISABLE_ELEMENT'
      }
      ,
      {
        element: 'ca3ChooseCredit28',
        state: 'DISABLE_ELEMENT'
      }
    ],
    data: [

      {
        id: 'ca37list_with_debitAccount37',
        type: 'List',

```

```
      value: {
          items: [
            {
              id: 1,
              value: 'test'
            },
            {
              id: 2,
              value: 'test'
            }
          ]
        }

    }
    ,
    {
      id: 'ca38selected_debitAccount38',
      type: 'Object',

      value: ''

    }
    ,
    {
      id: 'ca39list_with_creditAccount39',
      type: 'List',

      value: {
          items: [
            {
              id: 1,
              value: 'test'
            },
            {
              id: 2,
              value: 'test'
            }
          ]
        }

    }
    ,
    {
      id: 'ca40selected_creditAccount40',
      type: 'Object',

      value: ''

    }
    ,
    {
      id: 'ca41amount41',
      type: 'Money',

      value: ''

    }
  ],
  eventlisteners: [

```

```
  {
    element: 'ca5Confirm_Transaction32',
    value: 'DISABLE_ELEMENT',
    triggers: [
    {
      element: 'ca1create26',
      value: 'DISABLE_ELEMENT',
      type: 'ConcreteUI::InternalEvent'
    }
    ]
  }
,
  {
    element: 'ca1create26',
    value: 'ENABLE_ELEMENT',
    triggers: [
    {
      element: 'AccountAgreement',
      value: 'CALL',
      type: 'ConcreteUI::ExternalEvent'
    }
    ]
  }
,
  {
    element: 'AccountAgreement',
    value: 'CALLBACK',
    triggers: [
    {
      element: 'ca6select_debitAccount2',
      value: 'ENABLE_ELEMENT',
      type: 'ConcreteUI::InternalEvent'
    }
    ]
  }

    ],
    calls: [

  {
    element: "AccountAgreement",
    service: "",
    type: "QUERY",
    input: [],
    output: ["ca37list_with_debitAccount37"]
  }
,
  {
    element: "AccountAgreement",
    service: "",
    type: "QUERY",
    input: [],
    output: ["ca39list_with_creditAccount39"]
  }
,
  {
    element: "AccountAgreement",
    service: "",
    type: "QUERY",
    input: ["ca38selected_debitAccount38","
```

```
    ca40selected_creditAccount40","ca41amount41","
    ca42paymentDescription42","ca43generated_creditAccount43"],
153   output: []
154 }
155
156   ]
157 },
158 mutations: {
159  update: (state, payload) => {
160       let target = state.elements.find(e => e.element === payload.
    element)
161     console.info('[update] Processing (' + payload.element + ', '
    + payload.value + ')');
162
163     if(target == null) {
164       console.warn('Element ' + payload.element + ' added to store
    ');
165
166       let event = {
167         element: payload.element,
168         state: payload.value
169       }
170       state.elements.push(event);
171       Vue.set(state, 'elements', state.elements)
172     } else {
173       Vue.set(target, 'state', payload.value)
174     }
175   }
176 },
177 actions: {
178   update_state: (context, payload) => {
179     context.commit('update', payload)
180
181     context.state.eventlisteners.forEach(function(el) {
182         if(el.element === payload.element && el.value === payload.
    value) {
183           el.triggers.forEach(function(st) {
184               if(st.type == "ConcreteUI::InternalEvent") {
185                 context.dispatch('update_state', st)
186               } else if(st.type == "ConcreteUI::ExternalEvent") {
187                 context.dispatch('make_call', st)
188               }
189             }
190           )
191         }
192       })
193   },
194   update_data: (context, payload) => {
195     let target = context.state.data.find(d => d.id === payload.id)
196     if(target == null) {
197       console.warn('[data] Data does not exist')
198     } else {
199     var pattern = /(password)/gi
200       var value = payload.value
201
202       if(target.id.search(pattern)) {
203         //value.replace(pattern, '*')
204       }
205
206       console.info('[data] Data ' + payload.id + ' is set to ' +
```

```
    payload.value)
207         Vue.set(target, 'value', payload.value)
208       }
209     },
210     make_call: (context, payload) => {
211       context.state.calls.forEach(function(call) {
212         if(call.element === payload.element) {
213           if(call.type === "COMMAND") {
214             context.dispatch('command', call)
215           } else if (call.type === "QUERY") {
216             context.dispatch('query', call)
217           } else {
218             console.error('[system] Call can not be identified')
219           }
220         }
221       });
222     },
223     command: (context, call) => {
224       Vue.http.post('/service/' + call.service + "/" + call.element
    + "/").then(response => {
225         // JSON responses are automatically parsed.
226         let event = {
227           element: call.element,
228           value: "CALLBACK"
229         }
230         context.dispatch('update_state',event);
231
232         for (var d in response.data.bindings) {
233           const current = response.data.bindings[d];
234           let data = {
235             id: current.key,
236             value: current.value
237           }
238           context.dispatch('update_data', data)
239         }
240       })
241       .catch(e => {
242         console.log(e)
243       })
244     },
245     query: (context, call) => {
246       Vue.http.get('/service/' + call.service + "/" + call.element +
     "/").then(response => {
247         // JSON responses are automatically parsed.
248         let event = {
249           element: call.element,
250           value: "CALLBACK"
251         }
252         context.dispatch('update_state', event);
253
254         for (var d in response.data.bindings) {
255           const current = response.data.bindings[d];
256           let data = {
257             id: current.key,
258             value: current.value
259           }
260           context.dispatch('update_data', data)
261         }
262
263       })
```

```
      .catch(e => {
        console.log(e)
      })
    }
  },
  getters: {
    getState: (state) => (element) => {
      return state.elements.find(e => e.element === element)
    },
    getData: (state) => (id) => {
      return state.data.find(d => d.id === id)
    }
  }
})
```

Listing F.2: VueJS Store definition

```

import Vue from 'vue';
import VueResource from 'vue-resource';

Vue.use(VueResource);

let routes = [

  {

    method: 'GET',

    response:
    {
      bindings: [

        {
          key: "ca37list_with_debitAccount37",

            value: {
                items: [
                  {
                    id: 1,
                    value: 'test1'
                  },
                  {
                    id: 2,
                    value: 'test2'
                  },
                  {
                    id: 3,
                    value: 'test3'
                  },
                  {
                    id: 4,
                    value: 'test4'
                  }
                ]
              }

        }

      ]
```

```
    },
    url: '/service/AccountAgreement/'
  }
  ,
  {

    method: 'GET',

    response:
    {
      bindings: [

        {
          key: "ca39list_with_creditAccount39",

            value: {
                items: [
                  {
                    id: 1,
                    value: 'test1'
                  },
                  {
                    id: 2,
                    value: 'test2'
                  },
                  {
                    id: 3,
                    value: 'test3'
                  },
                  {
                    id: 4,
                    value: 'test4'
                  }
                ]
              }

        }

      ]
    },
    url: '/service/AccountAgreement/'
  }
  ,
  {

    method: 'GET',

    response:
    {
      bindings: [

      ]
    },
    url: '/service/AccountAgreement/'
  }

];

Vue.http.interceptors.unshift((request, next) => {
  let route = routes.find((item) => {
```

```
104     return (request.method === item.method && request.url === item.
        url);
105   });
106
107   if (!route) {
108   // we're just going to return a 404 here, since we don't want our
        test suite making a real HTTP request
109     next(request.respondWith({status: 404, statusText: 'Oh no! Not
        found!'}));
110   } else {
111     next(
112     request.respondWith(
113     route.response,
114     {status: 200}
115     )
116     );
117   }
118 });
```

Listing F.3: VueJS Interceptor definition

Appendix G

Proposal of a concrete syntax

To specify an instance of the task model we defined a proposal for a concrete syntax such that we can express task models in a textual form. The concrete syntax defines a simple language construct that enables the modeller to define different kinds of tasks. Listing G.1 shows the language construct of a task. Depending on the kind of task (abstract or interaction), the body of the task consists of an interaction class definition or subtasks. The task model can be recursively defined by nesting task constructs.

```
1 Task [Name] : [Abstract|Interaction|User|System] {
2   [Task | Behaviour]
3 }
```

Listing G.1: Simple language construct to define a task

Bibliography

[1] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software. Pearson Education, 2003.

[2] J. Vanderdonckt, A MDA-Compliant Environment for Developing User Interfaces of Information Systems, pp. 16–31. Springer Berlin Heidelberg, 2005.

[3] H. Ishii and B. Ullmer, "Tangible bits: Towards seamless interfaces between people, bits and atoms," in Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, CHI '97, (New York, NY, USA), pp. 234–241, ACM, 1997.

[4] P. Pinheiro da Silva, User Interface Declarative Models and Development Environments: A Survey, pp. 207–226. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.

[5] B. A. Myers and M. B. Rosson, "Survey on user interface programming," in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '92, (New York, NY, USA), pp. 195–202, ACM, 1992.

[6] J. Vanderdonckt and P. Berquin, "Towards a very large model-based approach for user interface development," 1999.

[7] R. L. Mace, G. e J. Hardie, and J. P. Place., "Accessible environments: Toward universal design.," in Design Intervention: Toward a More Humane Architecture, Center for Accessible Housing, North Carolina State University., 1990.

[8] S. Keates, P. J. Clarkson, L.-A. Harrison, and P. Robinson, "Towards a practical inclusive design approach," in Proceedings on the 2000 Conference on Universal Usability, CUU '00, (New York, NY, USA), pp. 45–52, ACM, 2000.

[9] P. A. Akiki, A. K. Bandara, and Y. Yu, "Adaptive model-driven user interface development systems," ACM Comput. Surv., vol. 47, pp. 9:1–9:33, May 2014.

[10] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt, "A unifying reference framework for multi-target user interfaces," INTERACTING WITH COMPUTERS, vol. 15, pp. 289–308, 2003.

[11] V. López-Jaquero, F. Montero, and P. González, T:XML: A Tool Supporting User Interface Model Transformation, pp. 241–256. Springer Berlin Heidelberg, 2011.

[12] OMG, "Mda guide," Specification revision 2.0, Object Management Group (OMG), 2014.

[13] B. Myers, S. E. Hudson, and R. Pausch, "Past, present, and future of user interface software tools," ACM Transactions on Computer-Human Interaction (TOCHI), vol. 7, no. 1, pp. 3–28, 2000.

[14] S. Wlaschin, Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#. The pragmatic programmers, Pragmatic Bookshelf, 2018.

[15] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero, "Usixml: a language supporting multi-path development of user interfaces," Ehci/Ds-Vis, vol. 3425, pp. 200–220, 2004.

[16] F. Paterno, C. Mancini, and S. Meniconi, ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models, pp. 362–369. Springer US, 1997.

[17] F. Paterno, C. Santoro, and L. D. Spano, "Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments," ACM Transactions on Computer-Human Interaction (TOCHI), vol. 16, no. 4, p. 19, 2009.

[18] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, N. Souchon, L. Bouillon, M. Florins, J. Vanderdonckt, et al., "Plasticity of user interfaces: A revisited reference framework," in In Task Models and Diagrams for User Interface Design, Citeseer, 2002.

[19] C. Phanouriou, UIML: A Device-Independent User Interface Markup Language. PhD thesis, Virginia State University, 2000.

[20] R. Schaefer, S. Bleul, and W. Mueller, "Dialog modeling for multiple devices and multiple interaction modalities," in Proceedings of the 5th International Conference on Task Models and Diagrams for Users Interface Design, TAMODIA'06, (Berlin, Heidelberg), pp. 39–53, Springer-Verlag, 2007.

[21] S. Feuerstack, M. Blumendorf, V. Schwartze, and S. Albayrak, "Model-based layout generation," in Proceedings of the Working Conference on Advanced Visual Interfaces, AVI '08, (New York, NY, USA), pp. 217–224, ACM, 2008.

[22] T. Clerckx, K. Luyten, and K. Coninx, DynaMo-AID: A Design Process and a Runtime Architecture for Dynamic Model-Based User Interface Development, pp. 77–95. Berlin, Heidelberg: Springer Berlin Heidel-

berg, 2005.

[23] K. Z. Gajos, D. S. Weld, and J. O. Wobbrock, "Automatically generating personalized user interfaces with supple," Artif. Intell., vol. 174, pp. 910–950, aug 2010.

[24] G. Lehmann, A. Rieger, M. Blumendorf, and S. Albayrak, "A 3-layer architecture for smart environment models," in 2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops), pp. 636–641, March 2010.

[25] M. Peissner, D. Häbe, D. Janssen, and T. Sellner, "Myui: Generating accessible user interfaces from multimodal design patterns," in Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '12, (New York, NY, USA), pp. 81–90, ACM, 2012.

[26] I. Kurtev, "State of the Art of QVT: A Model Transformation Language Standard," in Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers (A. Schürr, M. Nagl, and A. Zündorf, eds.), pp. 377–393, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. DOI: 10.1007/978-3-540-89020-1_26.

[27] O. M. G. (OMG), "Meta Object Facility (MOF) Specification (Version 1.4.1)," ISO Standard formal/05-05-05, Object Management Group (OMG), July 2005.

[28] A. Christoph, "Describing Horizontal Model Transformations with Graph Rewriting Rules," in Model Driven Architecture: European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004. Revised Selected Papers (U. A\s smann, M. Aksit, and A. Rensink, eds.), pp. 93–107, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. DOI: 10.1007/11538097_7.

[29] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering," in Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10, (Antwerp, Belgium), pp. 173–174, ACM, 2010.

[30] E. Kindler and R. Wagner, "Triple graph grammars: Concepts, extensions, implementations, and application scenarios," tech. rep., Technical Report tr-ri-07-284, University of Paderborn, 2007.

[31] "Jamda – java model driven architecture."

https://sourceforge.net/projects/jamda/.

[32] N. Skrypuch, "Jet - eclipse modeling - m2t — the eclipse foundation." https://www.eclipse.org/modeling/m2t/?project=jet.

[33] "Andromda." https://www.andromda.org/.

[34] D. H. Akehurst, B. Bordbar, M. J. Evans, W.G.J.Howells, and K.D.McDonald-Maier, "SiTra: Simple Transformations in Java," tech. rep., University of Kent, University of Birmingham, 2006.

[35] L. F. Pires, "Lecture notes in ADSA - Model Driven Engineering (2016-1b)."

[36] D. Akehurst and S. Kent, "A Relational Approach to Defining Transformations in a Metamodel," in UML 2002 — The Unified Modeling Language: Model Engineering, Concepts, and Tools 5th International Conference Dresden, Germany, September 30 – October 4, 2002 Proceedings (J.-M. Jézéquel, H. Hussmann, and S. Cook, eds.), pp. 243–258, Springer Berlin Heidelberg, 2002.

[37] B. Meyer, "On formalism in specifications," IEEE software, vol. 2, no. 1, p. 6, 1985.

[38] A. Hall and R. Chapman, "Correctness by construction: developing a commercial secure system," IEEE Software, vol. 19, pp. 18–25, Jan 2002.

[39] P. W. King, "Formalization of protocol engineering concepts," IEEE Transactions on Computers, vol. 40, pp. 387–403, Apr. 1991.

[40] G. Bernot, "Formal specifications and algebraic specifications," Proc of The 7th International Software Quality Week, vol. 17, p. 20, 1994.

[41] J. Stoel, "A case for Rebel, A DSL for Product Specifications." 2015.

[42] R. Chapman, "Correctness by construction: A manifesto for high integrity software," in Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software - Volume 55, SCS '05, (Darlinghurst, Australia, Australia), pp. 43–46, Australian Computer Society, Inc., 2006.

[43] J. T. Hackos and J. Redish, "User and task analysis for interface design," 1998.

[44] J. Annett, D. Cunningham, and P. Mathias-Jones, "A method for measuring team skills," Ergonomics, vol. 43, no. 8, pp. 1076–1094, 2000. PMID: 10975174.

[45] S. K. Card, The psychology of human-computer interaction. CRC Press, 2017.

[46] D. Kieras and M. Helander, "Toward a practical goms model methodology for user interface design," pp. 135–158, 01 1988.

[47] B. E. John and D. E. Kieras, "The goms family of user interface analysis techniques: Comparison and contrast," ACM Trans. Comput.-Hum. Interact., vol. 3, pp. 320–351, Dec. 1996.

[48] G. van der Veer, B. Lenting, and B. Bergevoet, "Gta groupware task analysis-modelling complexity," Acta psychologica, vol. 91, no. 3, pp. 297–322, 1996.

[49] F. Paterno, C. Mancini, and S. Meniconi, "ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models," in Human Computer Interaction INTERACT 97: IFIP TC13 International Conference on Human Computer Interaction, 14th to 18th July 1997, Sydney, Australia (S. Howard, J. Hammond, and G. Lindgaard, eds.), pp. 362–369, Springer US, 1997.

[50] D. Scapin and J. Bastien, "Analyse des tâches et aide ergonomique à la conception: l'approche mad*," Analyse et conception de l'IHM, pp. 85–116, 2001.

[51] R. J. K. Jacob, "Using formal specifications in the design of a human-computer interface," Commun. ACM, vol. 26, pp. 259–264, Apr. 1983.

[52] M. Tomasz and S. Gerd, "Modellierung sicherheitskritischer kommunikation in aufgabenmodellen (modelling safety-critical communication within task models)," Journal of Interactive Media, vol. 7, p. 39, 2017-07-26T15:22:37.783+02:00 2008.

[53] D. Zuehlke, K. Mukasa, A. Boedcher, and A. Reuther, "useml - a human-machine interface description language," Developing User Interfaces with XML: Advances on User Interface Description Language, 2004.

[54] H. Trætteberg, Model-based User Interface Design. PhD thesis, Norwegian University of Science and Technology, 2002.

[55] G. Botterweck, "A model-driven approach to the engineering of multiple user interfaces," in International Conference on Model Driven Engineering Languages and Systems, pp. 106–115, Springer, 2006.

[56] J.-S. Sottet, G. Calvary, J. Coutaz, and J.-M. Favre, "A model-driven engineering approach for the usability of plastic user interfaces," Engineering Interactive Systems, vol. 4940, pp. 140–157, 2008.

[57] M. Ben-Ari, "Mathematical logic for computer science," in Springer London, 1993.

[58] D. Zuehlke, K. Mukasa, A. Boedcher, and A. Reuther, "useML - A Human-Machine Interface Description Language," Developing

User Interfaces with XML: Advances on User Interface Description Language, 2004.

[59] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, vol. 45, pp. 1–17, USA, 2003.

[60] G. Taentzer, "AGG: A Graph Transformation Environment for Modeling and Validation of Software," in Applications of Graph Transformations with Industrial Relevance: Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers (J. L. Pfaltz, M. Nagl, and B. Böhlen, eds.), pp. 446–453, Springer Berlin Heidelberg, 2004. DOI: 10.1007/978-3-540-25959-6_35.

[61] F. Jouault, F. Allilaire, J. BÃ©zivin, and I. Kurtev, "Atl: A model transformation tool," Science of Computer Programming, vol. 72, no. 1, pp. 31 – 39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[62] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The epsilon transformation language," in Theory and Practice of Model Transformations (A. Vallecillo, J. Gray, and A. Pierantonio, eds.), (Berlin, Heidelberg), pp. 46–60, Springer Berlin Heidelberg, 2008.

[63] A. Rensink, "The groove simulator: AÂ tool for state space generation," in Applications of Graph Transformations with Industrial Relevance (J. L. Pfaltz, M. Nagl, and B. Böhlen, eds.), (Berlin, Heidelberg), pp. 479–485, Springer Berlin Heidelberg, 2004.

[64] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations," in Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I (D. C. Petriu, N. Rouquette, and O. Haugen, eds.), pp. 121–135, Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-16145-2_9.

[65] D. Varra and A. Balogh, "The model transformation language of the viatra2 framework," Science of Computer Programming, vol. 68, no. 3, pp. 214 – 234, 2007. Special Issue on Model Transformation.

[66] C. A. R. Hoare, "An axiomatic basis for computer programming," Commun. ACM, vol. 12, pp. 576–580, Oct. 1969.

[67] O. M. G. (OMG), "Object constraint language (ocl)," Tech. Rep. 1.3, Object Management Group (OMG), feb 2014. https://www.omg.org/spec/OCL/2.4/.

[68] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara, "Verification and validation of declarative model-to-model transformations through invariants," Journal of Systems and Software, vol. 83, no. 2, pp. 283 – 302, 2010. Computer Software and Applications.

[69] B. Bordbar and K. Anastasakis, "Uml2alloy: A tool for lightweight modelling of discrete event systems," in IADIS AC, 2005.

[70] D. Jackson, "Alloy: A lightweight object modelling notation," ACM Trans. Softw. Eng. Methodol., vol. 11, pp. 256–290, Apr. 2002.

[71] O. M. G. (OMG), "Mof query/view/transformation," Tech. Rep. 1.3, Object Management Group (OMG), jun 2016. https://www.omg.org/spec/QVT/1.3/.

[72] A. Schürr, "Specification of graph translators with triple graph grammars," in in Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching (D, Springer, 1995.

[73] M. Abed, D. Tabary, and C. Kolski, "Using formal specification techniques for the modelling of tasks and generation of hci specifications," The handbook of task analysis for human computer interaction, pp. 503–529, 2003.

[74] O. Sy, R. Bastide, P. Palanque, D. Le, and D. Navarre, "Petshop: a case tool for the petri net based specification and prototyping of corba systems," Petri Nets 2000, p. 78, 2000.

[75] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo, "Greatspn 1.7: graphical editor and analyzer for timed and stochastic petri nets," Performance evaluation, vol. 24, no. 1-2, pp. 47–68, 1995.

[76] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, and R. Valk, "An extensible editor and simulation engine for petri nets: Renew," in International Conference on Application and Theory of Petri Nets, pp. 484–493, Springer, 2004.