ulm university universität uulm

Ulm University | 89069 Ulm | Germany

Faculty of Engineering and Computer Science
Institute of Databases and Information Systems

Transformation of Activity-based Business Process Models to Complex User Interfaces: A Model-Driven Approach

Master Thesis Ulm University

Author:
Paul Hübner
paul.huebner@uni-ulm.de

Reviewers:
Prof. Dr. Manfred Reichert
Prof. Dr. Peter Dadam

Advisor:
Dipl.-Inf. Jens Kolb

2012

Version January 30, 2012

Danksagung - Acknowledgments

Mein Dank gilt allen Personen die mich bei der Erstellung dieser Arbeit unterstützt haben.

Besonderer Dank gilt Herrn Prof. Dr. Manfred Reichert, unter anderem für die Themenfindung und Vermittlung der vorliegende Arbeit. Des Weiteren gilt mein Dank meinem Betreuer Jens Kolb, der mich zielorientiert durch diese Masterarbeit geführt hat und der mich durch seine kritischen Anmerkungen stets vorantrieb.

Meiner Familie und Meiner Freundin danke ich für ihre Geduld und Unterstützung.

Zusammenfassung

Graphische Benutzeroberflächen sind eine wesentliche Voraussetzung für die Bearbeitung von Geschäftsprozessen. Hierbei stellen die Konzeption und Implementierungen graphischer Oberflächen einen Großteil des gesamten Entwicklungsaufwandes im Geschäftsprozessmanagement dar.

Um den Aufwand der Implementierung zu verringern, wird in der folgenden Arbeit ein Transformationsmodell zur ganzheitlichen und automatisierten Erzeugung graphischer Benutzeroberflächen, basierend auf Prozessmodellen, vorgestellt. Hierbei folgen alle Komponenten des Modelles dem *Separtaion-Of-Concerns* Grundsatz modellgetriebener Entwicklung. Die wichtigsten Komponenten bilden hierbei die *Transformation Patterns*, welche die Benutzeroberfläche generieren. Teil der *Transformation Patterns* sind die *Elementary Transformation Patterns* (ETP), diese erzeugen aus den einzelnen Prozessaktivitäten Interaktionselemente, welche durch die *Complex Transformation Patterns* (CTP) ergänzt werden. Diese erzeugen für komplexere Prozessmodellteile entsprechende Benutzeroberflächenelemente. Die Patterns werden durch einen Gruppierungsmechanismus ergänzt. Dieser ermöglicht unter anderem das Erzeugen von rollenspezifischen Benutzeroberflächen. Mittels einem *Mapping Meta Model* beschreibt das Transformationsmodell einen generischen Transformationsalgorithmus, zum ganzheitlichen erzeugen von komplexen Benutzeroberflächen. Zusätzlich erzeugt das Transformationsmodell eine bidirektionale Abbildung von Änderungen an der Benutzeroberfläche im Prozessmodell. Diese Fähigkeiten zur Abbildung von Änderungen ermöglicht wiederum das Umsetzten weiterführender Konzepte, wie beispielsweis die Anpassung einer Benutzeroberfläche zur Prozesslaufzeit und das nachträgliche Modifizieren des Prozesskontrollflusses.

Teile des Transformationsmodells wurde im Rahmen einer Machbarkeitsstudie implementiert. In dieser prototypischen Implementierung wurde die *Transformation Patterns*, der Gruppierungsmechanismus und der generischen Transformationsalgorithmus umgesetzt. Abschließend wurden in Tests, basierend auf vorher definierten unterschiedlichen Anwendungsfällen, komplexe rollenspezifische Benutzeroberflächen für komplette Prozessmodelle generiert, um so das Transformationsmodell zu verifizieren.

Abstract

An important but neglect aspect of Business Process Management (BPM) are the user interfaces required for human interactions during process execution. Moreover, user interfaces design and implementation requires a notable amount of all BPM related development efforts. Therefore in this thesis a model-driven approach for the overall automated generation of complex user interfaces for activity-based process models has been developed. A component-based Transformation Model is the core part of this approach. The components follow the separation of concerns principle of model-driven development.

The core component provides a hierarchical set of Transformation Patterns. Thereby Elementary Transformation Patterns (ETP) handle the aspects of user interface generation of single process activities. Supplemented by Complex Transformation Patterns (CTP) to transform complex process model fragments to user interfaces. These Patterns are supplemented with an advanced Grouping Mechanism, e.g., to enable the generation of role specific user interfaces. Through a Mapping Meta Model, the Transformation Model describes an overall generic Transformation Algorithm to generate complex user interface based on process models. In addition the Transformation Model enables a bidirectional propagation of user interface based changes to a process model and vice versa. These change propagation capabilities are the basis for more advanced features like user interface modification during the run-time of processes and retrospective modification of process control-flow.

As a proof of concept parts of the Transformation Model, more precisely, the Transformation Patterns, the grouping aspect and the overall Transformation Algorithm have been implemented in a prototype. Tests based on different use cases resulted in an overall automated approach for the generation of role-specific complex user interface for complete process models.

Contents

List of Figures

List of Tables

1 Introduction

The intensification of globalization in the last decades, forces the need for management of companies to revise their business processes [KLL09, vdAtHW03a]. Key success factors like the demand of quickly reacting on market changes, general higher productivity according to the overall product supply chain are only two examples, which indicate the need of using IT support for business processes to optimize them [vdAtHW03a, WSR09, DRRM^{+}10]. This requirements lead to the development of what is known as Business Process Management (BPM) which in short can be summarized as the management of all artifacts including the respective IT support, resources and people involved in business process execution [vdAtHW03a]. An important but neglect aspect for the execution of IT-based business processes are the user interfaces to support the required human interactions. These user interfaces take a notable amount of all BPM related efforts for their development [SMV10]. Especially, if business processes change it is common that the related user interfaces have to be re-implemented from scratch. This huge development effort for business process-related user interfaces is the starting point for this thesis.

1.1 Motivation

The acceptance and introduction of Business Process Management in companies has raised constantly during the last decade and is still a mayor key factor for successful business development in most areas [Wes07, Ham10]. In addition, the scientific community has discovered the various fields of BPM trying to address its different domains, with particular addiction to the involved formal methods [Dum05, vdAtHW03b]. In the course of this scientific movement, Van der Aalst et al. define Business Process Management (BPM) as *"methods, techniques and software to design enact, control and analyze operational processes involving human organizations, applications, documents and other source of information supporting business processes"* [vdAtHW03a].

The business processes, which are defined in process models capture the activities taking part during daily work in a company. These process models have to be executed to IT support BPM through the implementation and usage of so called *Business Process Management Systems* (BPMS). These are enterprise information system with direct support for modelling and execution of business processes models. Within this, the identification and definition of business processes is typically separated into two modelling levels. The first modelling level is used for the identification and functional definition of business process models and serves as an base for the second modelling level. On this second level, the functional process models are enriched with all, especially technical, details to deploy and execute the created business process models in the BPMS. This process models are also called technical process models.

BPM research and existing *Business Process Management Systems* address both modelling levels including the processes execution in a variety of details. A fundamental aspect during business process execution is the integration of user interactions and thus in consequence the user interfaces necessary. Since a lot of business processes are based on user interactions during the overall processing of business data, the involved user interfaces become a major part for efficient and user-friendly process execution [SSR+07]. Despite its significance in the context of BPM, the user interface creation is only handled to a certain extent.

A fundamental problem according to user interface creation for process execution is the overall semi-automated, in a lot of cases manual, approaches and thus the huge development efforts for implementing adequate user interfaces [SSR+07, ZZHM07]. An important aspect at this point is, when manually creating user interfaces, often a lot of process logic is captured implicitly in user interfactes instead explicitly in the process model. Thus, a distinct mapping between multiple activities in the process model and a user interface is not possible. This, in turn, results in additional manual efforts by adapting user interfaces if changes in the business process are necessary. No matter if the cause of these changes are based on the redesign of the business process model or the corresponding user interface [SMV10].

Since technical process models include detailed information which are necessary for user interface generation, e.g., executing agent of an activity and processed data, these models can be used as a base for user interface generation. This is the basic starting point for the approach followed in this thesis. Using technical process models as an input for the automated generation of agent role specific, complex user interfaces. With the additional effort to keep a well defined two-way relationship between particular business process models and user interface elements. Therefore a generic *business process to user interface Transformation Model* has been developed. The following chapter gives a clear and more detailed definition of the contribution by the developed Transformation Model.

1.2 Contribution

The contribution of the thesis is the development of an approach to generate user interfaces based on technical process models in an automated manner. The generated user interfaces are capable for the execution of the business process of the underlying process models. The developed approach links a business process model to a user interface and can be described in the three following steps:

1. **Prerequisite:** A block-oriented technical business process model, including detailed data flow definitions.
2. **Transformation:** Defined by the model-based approach developed in this thesis.
3. **Result:** User interfaces for handling the user interactions during business process execution.

In actual BPMS the user interface development is only addressed to certain extends [SMV10]. At least if more complex user interfaces, which are based on multiple process activities, should be created, at lot of manual development efforts are required. Therefore, the developed Transformation Model captures the user interface generation not only for multiple activities but also for complete process model instances. Moreover, it includes the possibility to allocate activities by grouping criteria. This results in a reduction of the development efforts for user interfaces and in complex user interfaces that are capable of processing multiple activities in a single step.

Another contribution of this thesis is to address the previously mentioned problem of losing the mapping between process model and user interface elements. Changes in process models may lead to additional changes to align the existing user interfaces. Alternatively, seen from a different perspective, at first the user interfaces are adopted to the new operational execution order and at second the underlying process models are aligned with the update user interfaces. This outlines the bidirectional manner of propagating changes in a BPMS. Independent of the chosen direction to implement the changes it is essential to know where, according to the process models and user interfaces, these changes have to take place.

Especially this should be addressed through this thesis resulting in an overall automated bidirectional mechanism for element mapping. The following listing outlines the two supported mapping directions:

1. Process model elements ⇒ user interface elements
2. User interface elements ⇒ process model elements

1.3 Organization of the Thesis

The general methodology throughout this thesis was based on an iterative course of action. In the following, the single most important parts are described as they are listed in the chapters of the thesis. The starting point has been a detailed study of research literature according to model-based user interface development and business process management related user interfaces. The results of this study are presented in Chapter 2. Further an analysis of two existing BPMS with special focus on their UI generation capabilities is provided in this chapter. In Chapter 3, some fundamental prerequisites according to process models and user interface models are outlined. Including the definition of a User Interface Model, which describes all required process model related user interface (UI) elements. In Chapter 4 requirements for complex user interface generation, based on uses cases and the related work are defined.

The user interface generation requirements are the basis for the Transformation Model which is introduced in Chapter 5. This Transformation Model, for user interface generation based on process models, is build up of *Transformation Patterns* and additional grouping capabilities. The Transformation Patterns describe how to instantiate elements of the previously defined User Interface Model. This, in turn, can be used for the generation of a user interface.

Chapter 6 starts with the discussion of additional grouping criteria of activities to take into account for user interface generation, e.g., organizational concerns to achieve role specific UIs. In the following, a Transformation Algorithm, which uses all parts of the Transformation Model is presented. Additionally this chapter discusses the aspects of delegating changes from user interfaces to process models and vice versa.

Chapter 7 covers process model run-time related aspects of user interface generation. These aspects include different granularity levels of UI-based changes during the run-time of a process instance and the retroactive modification of control-flow triggered by a generated complex user interface. The prototype which was developed as proof of concept during this thesis is presented in Chapter 8. It uses the concept of view generation for process models to implement the grouping aspects of the Transformation Model. For this view generation concept a the *proView* [RKBB12] prototype is extended. Finally, the thesis concludes with Chapter 9 summarizing the results of the previous chapters including some further research questions.

2 Related Work

Even though user interface are treated in a neglected manner in the area of Business Process Management System (BPMS) there exist some interesting research approaches, which should be discussed in this chapter. In addition, the general aspects of modeling user interactions and user interface, including their generation, should be introduced. Since these are important prerequisites for the approach of generating complex user interfaces for business process models.

Therefore this chapter starts with the presentation of well-establish task-oriented concepts according to UI modelling, supplemented with more business process related user interface research work in Chapter 2.1. In Chapter 2.2 two state-of-the-art BPMS are presented and analysed based on their user interface generation capabilities. Chapter 2.3 outlines the limitations of the presented approach according to an overall process model to user interface transformation.

2.1 Research Work

In general user interface (UI) development for IT systems is a various and complex field [HMZ11]. Research in the field of user interfaces has proposed model-driven development and thus the underlying modelling techniques to improve UI development. With the help of these modelling techniques, UI development should be simplified and more intuitive. In addition, such UI modelling techniques can be used for the creation of more formal definitions of UIs and the related requirements. In the field of model-driven user interface development there exist several modelling techniques for addressing different UI modeling related aspects [TKVW10]. One of them is the definition of user interactions by using task models [LV04]. Some of these task modelling techniques are of special interest according to the user interface generation for business process models, e.g., for the specification of required user interaction for process execution [KRW12]. Thus in the following some basics according to task models are introduce. These basics are supplemented by UI modelling and generation approaches for business process models.

2.1.1 Task Models

Basically, a *task*, can be described as an element to specify user interactions. A more general definition with respect to user interface development is given by Paternò [Pat00]. He defines a task as the steps necessary to reach a certain goal. In the field of Human Computer Interaction (HCI) this basic definition is shared by all reviewed task modelling approaches. In this sense task modelling is used to describe the interaction of a user in the context of model-driven user interface development [Szw11]. In their work for the comparison of different kind of task modelling approaches for user interface design Limbourg et al. [LV04] give a overview of relevant existing task modelling approaches. This overview includes fundamental HCI principles like the Goals, Operators, Methods, and Selection rules (GOMS) [CNM83] and Hierarchical Task Analysis (HTA) [AD67]. The most interesting task principle for our concerns according to UI generation is the so called ConcurTaskTree, (CTT) modelling notation [PMM97]. This is a mostly engineering-oriented approach, which uses the five concepts of tasks, objects, actions, operators and roles for the creation of models.

In ConcurTaskTrees the single tasks are arranged in a hierarchical tree. The root of the tree describes an abstract task which is refined by multiple sub-tasks contained on the next tree level. In addition, each task is of a certain kind (User, Abstract, Interaction, and Application) and is connected to adjacent tasks on the same hierarchy level by temporal relations. With the help of these temporal relations, it is possible to express the execution order of task, e.g. the necessity of parallel execution for two tasks. Figure 2.1 shows a sample CTT task model for an issue management application. This CTT is partitioned in three hierarchy levels and contains tasks for creating assigning and closing an issue. Additional to the user interaction tasks, it includes tasks for required system interactions like storing the data for an issue (details for this scenario can be found in 4.1.1).

The ConcurTaskTree approach has evolved towards a de facto standard for the description of user interactions in the field of model-based UI development [Szw11, GVC08, LV04]. Furthermore, there exist well-defined techniques to use CTTs as direct input for the generation of user interfaces [PS03, LVM$^+$05]. This leads to next chapter, which presents several approaches for the generation of user interfaces to support the execution of business processes.

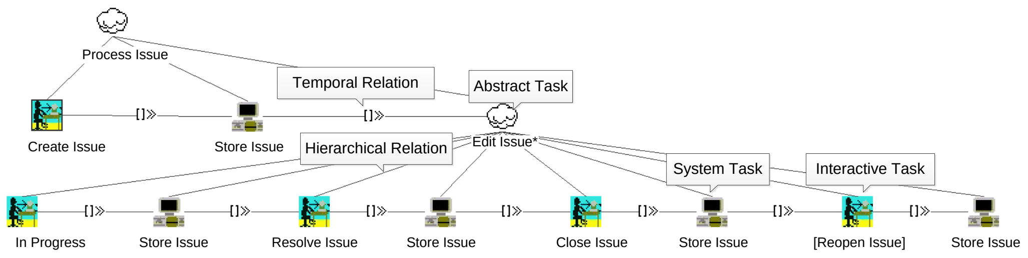


Figure 2.1: Issue Management Example ConcurTaskTree using CTT [MPS02]

2.1.2 User Interface Generation Approaches

FlowiXML [GVC08] is the implementation of a task model-oriented approach to support the generation of user interfaces for *workflow systems*. *Workflow systems* are predecessor of BPMS that mostly address the technical concerns of process execution. FlowiXML extends the UsiXML (USer Interface eXtensible Markup Language) UI modelling standard, which is based on task modelling, with additional capabilities required for workflow system user interfaces [LVM$^+$05]. In FlowiXML UIs are described by tasks, enriched with process model related characteristics like the description of workflow sequences and organisational concerns, e.g., assignment rules for tasks [RvdAtHE05]. The declarative task model of FlowiXML is based on ConcurTaskTrees. In it, XML notation is used to model a task. With the help of a generation tool these XML task descriptions are translate to a so-called abstract user interface (AUI), which, in turn is the input for the generation of a real user interface, e.g., based on web technology.

Sousa et al. use FlowiXML as technological input for their *Business Alignment Framework* [SMV10]. In this framework, a four-step approach is described in which a method is introduced to align the business process of an enterprise with the used information system as well as the user interfaces required for user interactions. In this Business Alignment Framework, the activities of a business process model are used as the basis for user interface generation. For each relevant activity, a CTT task model has to be created to describe a user interface on a conceptual level. Afterwards the ConcurTaskTree model is supplemented by a domain model [Ben96] which describes implementation concerns relevant for the connection of user interfaces with IT systems. Based the ConcurTaskTree model and the domain model FlowiXML generates a abstract user interface which in a final step is transformed to concrete user interfaces for the execution of business processes [Sou09].

An interesting addition of this Business Alignment Framework is the possibility to track change requirements between user interfaces and business process models in a bidirectional manner. This is realized by the definition of rules for the direct transformation of process model elements to task model elements and rules which define changes in the process model based on changes in the task model, e.g., adding a new task would result in adding a new activity in the process model [SMV10].

Guerrero et al. define a approach for the transformation of workflows, which they define as the technical realization of a business processes, to user interfaces [GVGW08]. The approach also uses CTT for task modelling and is to a certain extend similar to the approach of Sousa et al. [SMV10]. Differences are that Guerrero et al. address more technical and implementation related concerns but omit the tracking of changes between UIs and process models.

Zhao et al. describe an approach for the specific use case of generating user interfaces based on business process definitions for e-commerce systems [ZZHM07]. They use process model data, e.g., like the names of single activities, and transformation rules to map the extracted data to certain task-oriented user interface transformation rules. For example, if an activity is named *search product* it is mapped to a search task, for which, in turn, a specific UI to perform a search is generated

The *PHILharmonic Flows* research project includes an example for a different paradigm to model and generate user interfaces for business process execution [KR11b]. This approach is based on the data object-oriented view of business processes. Therefore a business processes is described as the changing states of data objects within process execution and the relation between this changing data objects. Supplemented with user specific access rules for each data object and transformation rules for data types, role specific user interfaces for execution processes can be generated [KR11a].

The limitations of the presented user interface generation approaches and the challenges for an overall complex user interface generation are outlined in Chapter 2.3. The following chapter analyses user interface generation as supported by actual BPMS.

2.2 Tool Support

In the following, the principles of user interface generation by two actual BPMS are analyzed. Starting with AristaFlow[1], which is based on ADEPT [Rei00] project and offers user interface generation options for process model activities. IBM WebSphere Lombardi Edition[2], is the second BPMS which is analyzed according to its user interface generation capabilities. It offers the feature to model processes on two hierarchical levels, which are interconnected with each other. Based on this, the second hierarchy level is used to model the basic course of action inside user interface forms.

2.2.1 AristaFlow BPM Suite

The AristaFlow BPM Suite offers capabilities for the realization of a complete process-oriented IT environment. Since we want to focus on the particular aspects of user interface generation in the following, this AristaFlow aspects are introduced. An important component of AristaFlow is the Process Template Editor. It is used for the graphical modelling of processes and to connect the process models with required IT Systems for their execution. In addition, the Process Template Editor offers the functionality required for user interface creation. A basic concept of AristaFlow is the separation of its graph-based process model from the actions that have to be performed in the nodes (activities) of this model. The concept for this abstraction mechanism is called Activity Template and implemented as defined by the ADEPT process model [Rei00].

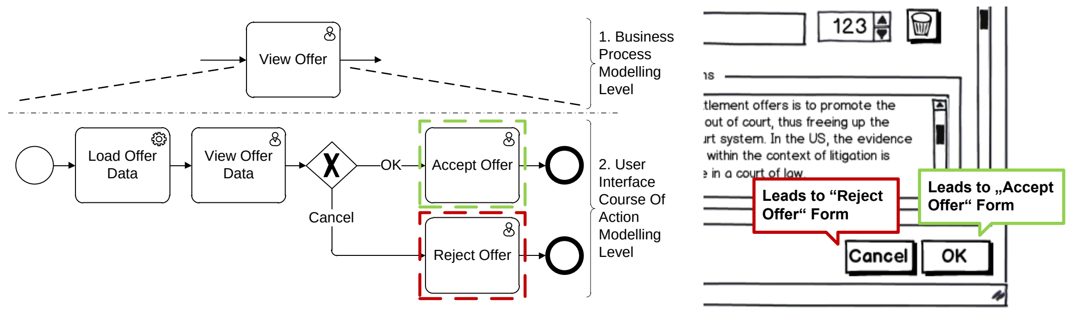
By the use of Activity Templates, it is possible to assign different kind of actions to nodes (activities) of a process model, e.g., there is an Activity Template for fetching data from a previously defined database. The Activity Template mechanism is also used if for a certain node a user interface form should be generated. The only thing to do while creating a process model is to assign the respective Activity Template (*Generated Form or User Form*) to a node. During the run-time of process instances for each of the nodes with an assigned, Activity Template for user interface generation a simple form is generated. This form is based on the data-flow of the node, which has to be defined in advance. In addition, it is possible to remove single data elements to omit them during form element generation. Summing up the user interface generation in AristaFlow consist of the following four steps

[1]`http://www.aristaflow.com/AristaFlow_BPM-Suite.html` (in German), last checked January 30, 2012
[2]`http://www.ibm.com/software/integration/lombardi-edition/`, last checked January 30, 2012

1. Process model creation, including the detailed definition of the respective data-flow.
2. User Interface form specific Activity Template assignments to process model nodes, which require user interaction.
3. Individual Modifications (removing/adding) of the single data elements considered for user interface form generation.
4. Standardized user interface generation during process run-time based on the process instance data for all process model nodes with appropriate assigned Activity Templates taking the previously user defined data element modifications into account.

2.2.2 IBM WebSphere Lombardi Edition

IBM WebSphere Lombardi Edition is based on IBM WebSphere Server and the integration of the Lombardi TeamWorks BPMS after the acquisition of Lombarid by IBM in 2010 [YSW+10]. Further, it offers a wide range of BPM functionality including extended capabilities according to process modelling and user interface generation.


View Offer
1. Business Process Modelling Level
Load Offer Data
View Offer Data
OK
Cancel
Accept Offer
Reject Offer
2. User Interface Course Of Action Modelling Level
123
tlement offers is to promote the
out of court, thus freeing up the
ırt system. In the US, the evidence
within the context of litigation is
in a court of law
Leads to "Reject Offer" Form
Leads to „Accept Offer" Form
Cancel
OK

(a) IBM WebSphere Lombardi Edition two Process Model Levels (b) Resulting User Interface

Figure 2.2: User Interface Modelling in IBM WebSphere Lombardi Edition

Figure 2.2 shows a conceptual example for the hierarchical process model creation in the IBM Lombardi Authoring Environment. The process models are defined on two connected modelling levels. The first of these levels is used for the definition of global process models whereas the second level can be used for the definition of required fine-grained actions. In the case of user interface generation, this second modelling level can be used for more technical process models as they are needed for the processing of user interfaces during the run-time of a process. Figure 2.2a shows an example for such a user interface processing related workflow. This includes the subsequent user interface related actions to perform for the different kind of processing options provided by the offer form of Figure 2.2b. As shown in the example depending on the action performed by a user during process run-time, either an subsequent accept offer form (user clicks on *OK*) or an reject offer form (user clicks on *cancel*) is displayed.

The user interface generation is triggered by assigning *Coaches* to the activities on the second hierarchy modelling level. These Coaches trigger a user interface generation, which uses a previously modelled data-flow for the creation of single form elements, similar like in the Activity Template approach of AristaFlow (cf. Chapter 2.2.1). Another addition is the possibility for arrangements of elementary form elements with help of an graphical user interface editor. As shown in Figure 2.2 during process run-time the previously modelled user interfaces internal workflows are implemented by the process engine. For the presented sample case this would include the displaying and processing of two different forms if the activity *View Offer* is executed. At first a form which shows the data of an offer is displayed followed by either an accept offer form or a reject offer form depending on the action performed by the executing user.

2.3 Limitations & Challenges

In the following previously presented research work and Business Process Management Systems are reviewed according to an overall complex user interface generation. Additionally the concepts of model based user interface derivation with special focus on task modelling are analyzed.

Research work which discusses business process model related user interface concerns like [GVGW08, SMV10] presented in Chapter 2.1.2 focuses on more general principles of user interface generation instead of concerning the combination of multiple process model parts for user interface generation. The focus of task-oriented approaches is a more detailed interaction description, which should result in user interface with better usability and in turn enable a more effective process execution. The tracking of changes between business process and the user interface required for their execution as described by Sousa et al. is an interesting aspect [SMV10, SV11]. This enables the detection of user interface artifacts, which are concerned form changes in business process models and vice versa.

The approach of FlowiXML mainly covers the generation of declarative UIs by using task models but also does not cover the consideration of multiple activities for one screen element [GVC08]. This is what all previous research seems to have in common, a complex user interface which covers multiple process model activities to summarize them in a single screen element, has not been described yet.

The de facto situation for existing Business Process Management Systems is very similar. For advanced more complex requirements, for user interface related concerns additional manual development efforts are necessary. This is also the case for the presented AristaFlow BPM Suite. The integration of complex user interfaces is possible but requires significant manual development efforts. A supporting approach for their automated generation is missing.

IBM WebSphere Lombardi Edition offers possibilities to model more complex user interface related concerns with help of its two level-separated process-modelling environment. This approach is to a certain extend similar to the task modeling research approaches. The second modelling level can be used to specify user interface interaction details, which is also the basic intent of using task models. However, the IBM WebSphere Lombardi Edition modelling environment is rather complex and if the interaction details necessary for a user interface are more elaborate as in the presented example, additional manual development efforts are required.

Since the popularity of task models for user interface generation in the following an analyze of how user interfaces can be derivated from such task models is presented. This is supplemented by some general remarks of actions required to obtain user interface elements from model based UI descriptions. As mentioned before a task is a description mechanism for user interactions. At which a task consist of the steps required to reach a certain goal [Pat00]. Since this is a rather abstract description, a well-defined task environment is required to use such a description for the generation of user interfaces from (e.g. UsiXML [LVM$^+$05]). As outline in the related work chapter, UsiXML uses XML and multiple connected context levels to generate a user interface description based on tasks.

Table 2.1 shows a comparison of the user interface generation approaches previously introduce. For more details an enlarged version of this table can be found in appendix A.1 (Table A.1). The presented approaches are derived into conceptual- and implementation steps that have to be performed to obtain a user interface. The task model based user interface generation of the *UI Business Alignment* approach by Sousa et al. is based on UsiXML. It can be seen as an adoption for UsiXML to generate user interface for business processes.

		UI Bussines Alignment (Sousa et al.)			Philharmonic Flows (Künzle et al.)		ConcurTaskTrees (Paterno et al.)	
Conceptual Steps	1.	List of Mapping Busines Process to Task Model Elements			1.	derivate hierarchical process models from data object processing in information systems by :	1.	Decied How to Create Presentations for Tasks (1. Same for All Task Types, 2. Different for all Task Types, Combination of 1. & 2.)
		Basend on BPMN , Separated into Mappings for:				1. Macro Processes = Data Object Interaction	2.	Identifiy which Task can be grouped in one User Interface by checking which Tasks can be enabeld at the same time
		1. Bussines Elements to Task Elements						
		2. Activity Attributes to Taks Properties				2. Micro Processes = Data Object Behvaior	3.	Decied How to Arrange the Functionality of theses Tasks in a UI by considering Data and Control Flow Connections
		3. Process Activities and Task Types						
	2.	Four Development Steps:		Based on UsiXML / FlowiXML	2.	Prerequisite: Detailed Data Type Model	4.	Validate the Set of enabled Tasks about Contained Task Patterns
		1. Creat Conceptual Models (Task Models, Data Models, User Models)				For Each Data Type: define a Micro Process (= finite state machine)	5.	Choosing Presentation Template based on Task Semantic
		2. Create Abstract UI (AUI) using UsiXML and the Conceptual Models				Connection of DataInstances (Micro Proecesses) = Macro Process	6.	Considering Temporal Relation between Tasks (This Seems to be a good Input for the CTPs (cf. Paterno Model Based Design Book p 82.)
		3. Create Concret UI (CUI) based on AUI				Rolse specific Acces on Macro & Micro Level defined in a Authorisation Tabel	7.	Processing The connections between multiple Enabled Task Sets (= CTP Pattern nesting & Block Processing)
		4. Create Final UI (FUI) based on CUI (e.g. html, swing etc.)				Authorisation Tabel is used to generate Role Specific Forms, by using data type & data access based transformation rules	8.	Ordering UI Elements for Processed Data based on their priority (level in the cct)
Implementation	BP Model	BPMN			No information about the concret steps taken could be found. The basic principal is as described in the last conceptual step:		1.	Use Arichtectural Model to describe UI Components (Consists of Several Sub Models e.g. MVC, or model to describe interactions)
	Task Model	ConcurTaskTrees (Global Task Model Based on Complete BP Model, with Sub Task Models based on BP Activities)						
	Abstract UI	User Interface without Representation Form based on CTT Model	UsiXML (CAMELEON Reference Framework default Implementation), FlowiXML process/ Workflow Extension for UsiXML		The Authorisation Table is ued to gerate Role Specific Forms for the processing of data objects. To achieve this data type and data access based transformation rules are applied		2.	Transformation From Task Model to UI Architekture Model = Map Tasks to SW UI Objects (interactors) cf. Paterno Model Based Design Book p. 115 R1) -R3)
	Concret UI	AUI + Representation Form (default form is 'graphical'), Plattform indetent		OWLAPI (OWL Ontology Processing); SWRL Bridge (Transform SWRL Rules in OWL); Drools (Manage Models)			3.	Connect the Interactors to support Information Flow, cf. Paterno Model Based Design Book p, 116 R4) - R19)
	Final UI	Implementation of CUI by transformation to e.g. HTML					4.	Task Patterns describe Common UI Usecases e.g. "search", Theses UsesCases imply a certain Task Structure connected with a certain UI design

Table 2.1: From Model to User Interface Elements, Approach Comparison

In the *PHILharmonic Flows* approach description of how to generate user interface is only available at a conceptual level. Any explanatory notes about implementation technology related things are missing. This is similar for the CTT approach with the difference that this primarily was develop as a conceptual approach [PMM97].

The general problem in the overall UI generation approaches is the missing of concrete required actions to transform specific model elements to specific user interface elements (UI Widgets). These are often only described on a conceptual level or if implementation specific only vague formulated. This results in a limitation of mapping a task model to a user interface and vice versa since many different resulting UIs derived from a task model are possible [SV11].

The special focus of task based user interface modelling are usability aspects. Task modelling tries to create a hierarchical, task oriented description of all execution sequences a user requires to perform needed actions. This is somehow in contrast to the approach of this thesis, which tries to describe a standardized complex user interface that enables a user to perform the actions required for the execution of a business process. In consequence, usability aspects are only treated subordinate. In the following Chapter, the fundamentals required for a complex user interface generation approach, are presented. This includes process and user interface modelling definitions.

3 Fundamentals

In this chapter, the fundamentals required for the development of an automated transformation mechanism from business process models to complex end user interfaces are introduced. These fundamentals are separated into two parts. In Chapter 3.1, all required definitions according to BPM, mainly considering process modelling, are specified. Chapter 3.2 outlines all user interface related prerequisite with a special focus on automated generation of user interfaces. In addition the so-called *User Interface Model* that offers a model-based declaration mechanism for user interface elements, used and referenced in subsequent, is introduced.

3.1 Business Process Management

In the following BPM concepts required for the description of a user interface generation approach are introduced. This includes the definition of a process model specification. For the definition of business process models there exist several different notations. In their Business Process Management (BPM) standards overview paper Ryan et al. introduce popular business process modeling approaches including a classification of them [KLL09] . The classification is based on the purpose of such a modeling standard. Whereas the authors distinguish between graphical, execution, interchange or diagnostic intended process-modeling standards. As an example in this classification BPMN [OMG11a] would be categorized as a graphical modeling standard whereas BPEL [OAS07] would be categorized as execution standard. For our purpose of user interface generation, we want to focus on executable process models. Since only if a business process can be executed, there is a need of user interaction and thus user interfaces are required for process execution.

The ADEPT[1] [Rei00, DR09, DRRM^{+}10] process model fits ideal for the needs of user interface generation. It has a block structured executable process model notation with clear

[1]The name ADEPT is a acronym for *Application DEvelopment based on pre-modeled Process Templates* [Rei00], but is used self-contained by now.

semantic definitions. It offers features like correctness by construction, advanced process run-time capabilities including process (model-) ad-hoc changes.

In the following the required process model elements for build block-oriented process models which serve as input for complex user interface generation are described. For this purpose Business Process Model and Notation (BPMN) is used as graphical process modeling notation [OMG11a].

3.1.1 Block-oriented Process Models

Before defining the single process model elements used subsequently a definition of the term *block-oriented* is given. In their work about guidelines for reducing process model complexity, La Rosa et al. introduce the block-oriented process model paradigm as most important guideline to make a process model understandable [LRWM$^+$11]. For their pattern based definition of this guideline they refer to the *Seven process modeling guidelines* in which block-oriented modeling was denoted as important aspect for an intuitive understanding of process models [MRvdA10].

In this sense in a block-oriented process model each element, which splits the control-flow to multiple possible paths, requires a corresponding join element for merging the paths again. The type of the join and merge element have to be the same. In addition, it is possible to nest such join and merge pairs (blocks) in a proper way. Whereas proper refers to the fact that an intersection between blocks, which have a different nesting level, is not allowed.

The core for all subsequent process model elements and the definition of their control-flow is based on an attributed directed graph [Rei00]. The nodes of this graph represent the activities of the process model. At which the edges between the nodes represent the control-flow. This graph and the respective nodes are arranged in a block-oriented manner. Additionally the graph has a single start and end node and each node (expect the start node) has at least one predecessor and one successor (expect the end node).

Figure 3.1 shows a catalog of primitive graphical process model elements (using BPMN) that are used for the definition of all required advanced block-oriented constructs. The primitive elements and their semantics are introduced in the following. The first element *Human Activity* represents an activity (node in the graph) which has to be executed by a human resource. Followed by the *Service Activity* element which refers to all system-oriented activities, e.g. database operation or service calls to any kind of IT-System. The third element is *Subprocess Activity* this is a placeholder for a complete process model, it is used to reduce complexity of single process models and to express process model modularity. In this sense the fourth and last activity element *Loop Activity* has similar semantic as the Subprocess Activity. The difference is that it additionally expresses the requirement that the underlying activity elements are executed multiple times.

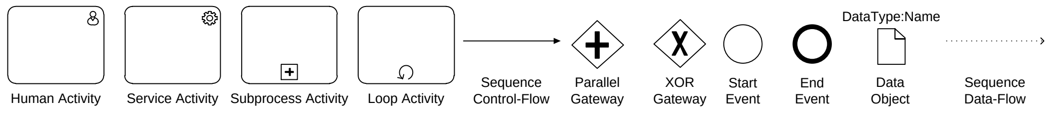


Figure 3.1: Catalog of Primitive Process Model Elements using BPMN [OMG11a] notation

The next element *Sequence Control-Flow* represents an edge in the process graph. It is used to connect single activities with each other. The *Parallel Gateway* is a special node which splits the Sequence Control-Flow in multiple parallel sequences. Based on the block-oriented process model paradigm each splitting parallel gateway requires a joining parallel gateway. The *XOR Gateway* also defines the splitting of the Sequence Control-Flow to multiple sequences. Based on a exclusive decision only one of this multiple control-flow splitting sequence is executed. The *Start Event* and the *End Event* element represent the respective start and end nodes in a process graph.

The last two process model elements to introduce differ from the previous elements since they do not directly refer to the process model control-flow. The *Data Object* element is used for modelling the data elements required in a process model. The Data Object can be connected with any kind of the previously presented activity or gateway elements using the *Sequence Data-Flow* element. For the Data Object the textual notation *Datatype:Name* on top of the graphical element defines the data type and the name of the Data Object. The data type can either be primitive (Integer, Float, Boolean, String, Date and URI) or complex (use case specific data objects).

Additionally the direction of the dotted edge for moddeling the Sequence Data-Flow indicates by an arrow in which manner a Data Object element is accessed by an activity. Figure 3.2 illustrates the different possible characteristic of data access. If the arrow points from the data object to the activity, the activity reads the data (cf. Figure 3.2a). Whereas if the arrow points from the activity to the Data Object the activity writes the data (cf. Figure 3.2b). A combination of those two options results in an activity editing the data (cf. Figure 3.2c). The modelling of process model data-flow is essential for the presented user

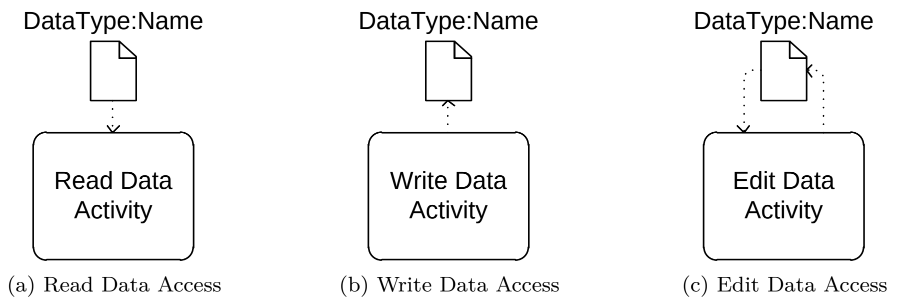


Figure 3.2: Different Types of Data Access Modelled with BPMN

interface generation approach. Thus, more particular details about data types and access are described in Chapter 5.3, which deals with the handling of activity data-flow in the case of user interface generation.

With the help of the introduced primitive process model elements now the more advanced elements are introduced. As mentioned before these element are based on the ADEPT process model [Rei00]. The most important of these process model elements is the *Control-Flow Block*. It is the base element to build a block-oriented process model and consist of a set of nodes and edges. Additionally a Control-Flow Block always consists of an opening and closing primitive process model element (node) and can be nested with other Control-Flow Blocks [LRWM+11]. Control-Flow Blocks can be defined by using the primitive process model notation elements introduced before. In the following a set of five Control-Flow Blocks with different semantics are introduced.

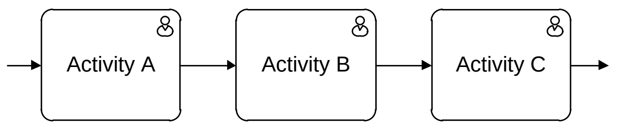


Figure 3.3: Exmaple Sequence Control-Flow Block

The most basic Control-Flow Block (CFB) is the *Sequence CFB*. By definition it arranges activities or other Control-Flow Blocks in sequence with each other. Figure 3.3 shows an example for such a Sequence CFB. In this the three activities (Activity A, Activity B, Activity C) are executed in sequential manner starting with Activity A followed by Activity B etc.

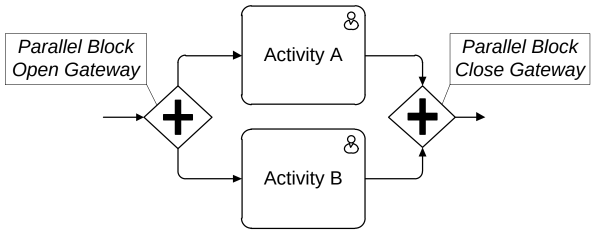


Figure 3.4: Exmaple Parallel Control-Flow Block

Figure 3.4 shows an example *Parallel CFB*. A Parallel CFB consists of a opening and closing parallel gateway and of a set of activities or different Control-Flow Blocks which are executed in parallel. Referring to the example of 3.4 this indicates Activty A and Activty B are executed simultaneously in parallel order.

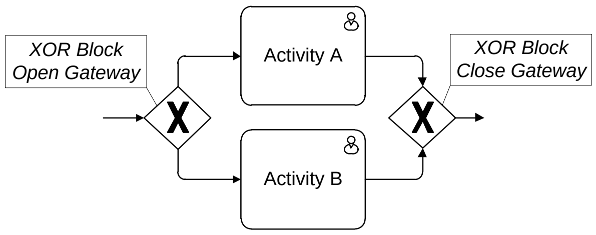


Figure 3.5: XOR (exclusive choice) Control-Flow Block

The *XOR CFB* also consists of an opening and closing gateway but of the type XOR (exclusive choice). And a set of activities or different control-flow blocks. Figure 3.5 shows an example for an XOR Control-Flow block. In addition, the blocks contained in a XOR CFB are assigned to a set of, at least two, execution paths. The starting XOR Gateway decides by data input which of the execution paths is activated and executed.

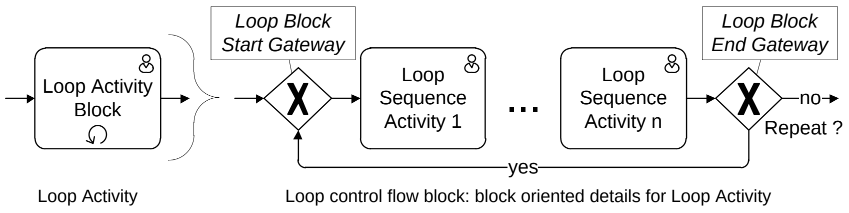


Figure 3.6: Example Loop Control-Flow Block

A *Loop CFB* as shown in Figure 3.6 is used to describe the iterative execution of a set of Activities or Control-Flow Blocks. For an informal process model, a single Loop Activity element can be used to model a loop. According a block-oriented definition the left-sided loop notation of Figure 3.6 is required. This includes a start and end Loop Block Gateway to define which nested Activities are executed iteratively. Additionally the Loop Block End Gateway requires data input to decide about quitting the Loop CFB.

The last Control-Flow Block is the *Subprocess CFB* this a special marker or placeholder Control-Flow Block linking to a different process model which has to be executed if the Subprocess Control-Flow Block is reached during process execution.

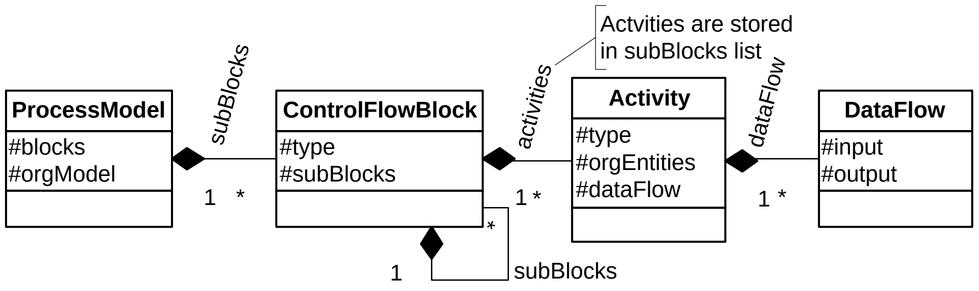


Figure 3.7: Process Model Element Hierarchy (as UML Object diagram) relevant for User Interface Generation based on ADEPT [Rei00]

With the help of these different Control-Flow Blocks now a more specific definition of a process model, which will be used for user interface generation, can be given. In this sense a process model consist of a basic Sequence Control-Flow Block. This basic Sequence Control-Flow Block consists of arbitrary activities or different Control-Flow Blocks.

An additional aspect to mention for process models, if referring to user interface generation concerns, is the *Organizational Model* used in those process models. An Organizational Model can be summarized as the definition of roles, functions and positions for human resources in a organization [CT12]. Such an Organizational Model is in turn used to allocate the activities of a process model to specific human resources. Figure 3.7 summarizes the user interface generation relevant aspects of the block-oriented process model definition in a hierarchical UML Object diagram.

3.1.2 Process Life Cycle

The so called *Process Life Cycle* describes the development and execution of business process models as cyclic phases [WSR09, HBR08]. In [WSR09] the four cyclic phases are defined as illustrated in Figure 3.8. The *design* phase is defined as the phase in which business

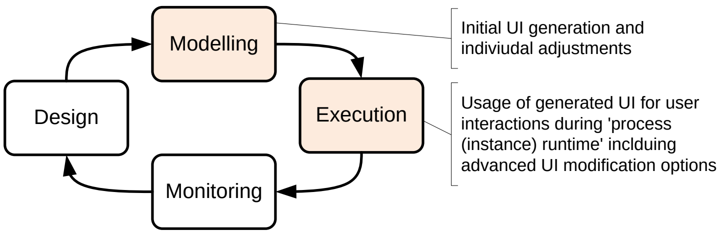

Modelling
Design
Execution
Monitoring
Initial UI generation and indiviudal adjustments
Usage of generated UI for user interactions during 'process (instance) runtime' inclduing advanced UI modification options

Figure 3.8: Process Life Cycle [WSR09], with User Interface Generation relevant phases highlighted

processes are developed on an abstract management-oriented level. In the *modelling* phase these abstract process models are enriched with execution information and transformed into an executable definition like Petri nets or ADEPT. These technical, executable process models are the entry point for the introduced complex user interface generation approach. In the *execution* phase the previously defined process models are instantiated. Thus, the generated user interfaces are used for human interactions during the process execution. The last phase in the process life cycle is the *monitoring* phase. There post execution analysis is performed.

This leads to two phases in the process life cycle, which are relevant for user interface generation:

1. Process **modelling** phase. In this phase, a customizable user interface preview can be generated. This preview of the of a user interface during process run-time can enable an individual adoption of single UI elements or even modifications of the complete look of the UI. The single steps necessary for this are described in the Chapter 5 and Chapter 6.

2. Process **execution** phase. In this phase, the generated user interface has to be connected with the process execution engine and instantiated with run-time data. With the help of the ADEPT, it is possible to enable UI-based modifications of the process during its run-time. This might result in actions like adding the UI elements for a new activity to add this new activity in the process model instance. Additionally advanced process control-flow modifications [RDB03], like stepping backwards and choosing different options resulting in a different execution path, will be enabled by generated complex user interfaces. The process run-time related aspects of complex user interfaces are presented in Chapter 7

With the help of the block-oriented process model specifications including their graphical notation and the user interface generation related requirements according to process life cycle phase, now it is possible to describe the transformation of such a process model to a user interface. Before describing this process model to user interface transformation some general remarks and definitions about user interface and their model-based development are required. These are outlined in the following.

3.2 User Interface Models

This Chapter is intended to give a brief introduction to model driven user interface development since the UI generation approach developed in this thesis is based on model-driven engineering techniques. Therefore, in Chapter 3.2.1 some general concerns according to model driven user interface development are presented. Followed by Chapter 3.2.2 in which the User Interface Model used for the process model to user interface generation is introduced.

3.2.1 Model-Driven User Interface Development

Model-Driven Development (MDD) in general is an important way of developing software with the help of models [HMZ11]. In the sense of MDD a model can be seen as abstract representation of certain system aspects. A model is often either represented visual e.g. by using Unified Modeling Language (UML) [OMG11b] or in a textual form e.g. by using Extensible Markup Language (XML) [WWWC08] or a combination of both. In MDD oriented software development all involved development artifacts are called a model [PEGM94]. Model transformation is used to transform a model into another less abstract model and finally into implementation code. MDD approaches now provide the concepts and tools to deal with such model artifacts [HMZ11].

Model-Driven User Interfaces development applies the principles of MDD to the domain of UI development. A lot of approaches for model-based user interface development have been proposed since the emerge of model-based development in the 1990s [PEGM94, TMN04, XJ07]. However, a widely accepted standard for model-based UI development is nowadays still out of reach but there exist several competing approaches [XJ07, PM97, GVC08]. Some of them mostly for specific use cases and domains seem to work well and are accepted in their area [GBP$^+$01]. In general, UI development is still a time consuming complex manual development task [LW07].

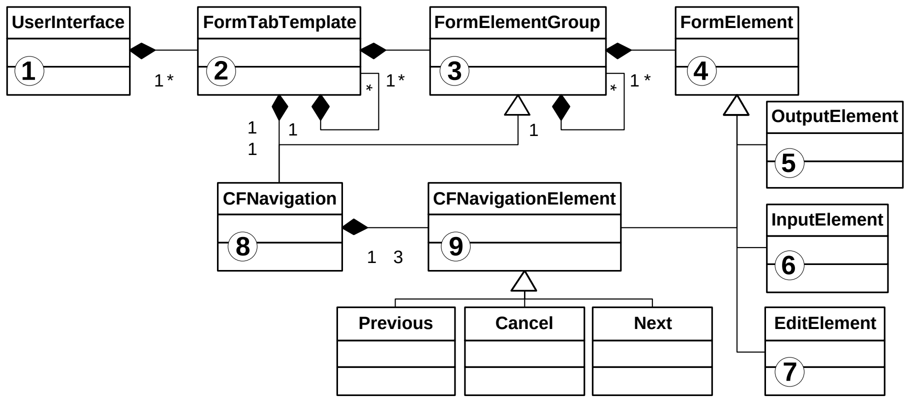
The basic principles of model-driven user interface development are directly adopted from MDD. The first of them to mention is the separation of different concerns in different models. In which the so called task modelling for the definition of user interactions in a more or less abstract form is an important aspect (cf. Chapter 2.1.1). The second important aspect is the hiding of UI implementation specific technical details. Since modern UI development relies on a huge pre-assembled software stack build upon various frameworks and libraries and is additionally often a computing platform specific task [HMZ11]. The third and last important aspect to mention is the purpose of separating the content respectively the data from the design (how the data is displayed).

To achieve this content and design separation the concept of domain models for user interfaces has been developed [Ben96]. Using this, it is possible to extend UI development with capabilities for flexible adaptable look and feel of generated user interfaces. This, in turn, results in domain specific user interfaces that can fulfill established usability specific requirements, e.g., by adopting, cooperate design and styleguides in this domain model. After this short model-driven user interface development overview the following chapter presents the User Interface Model developed to realize a business process model to user interface transformation.

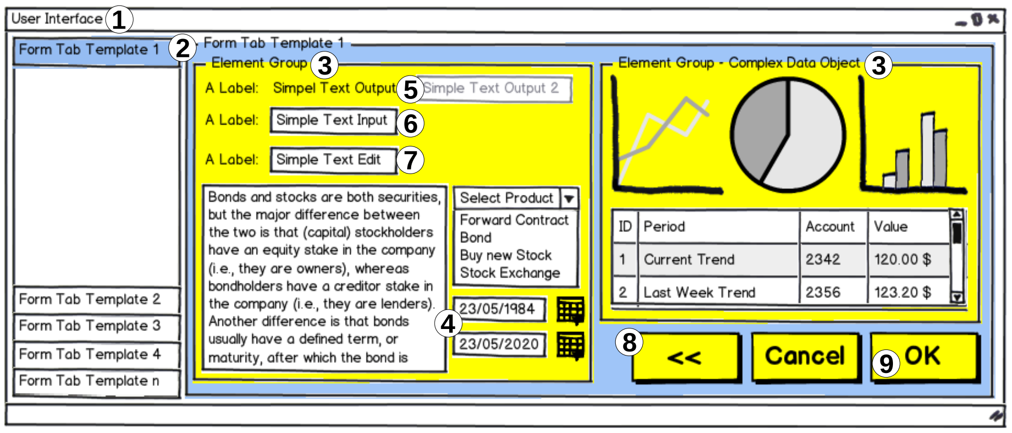
3.2.2 Transformation User Interface Model

The *User Interface Model* (UI Model) presented in the following is described by a hierarchical data structure that describes display elements which are used in user interfaces for the execution of process instances. In the sense of model-driven development, it is used for separating the concerns of execution a process workflow from a graphical representation required for necessary user interactions. With help of the UI Model, it is possible to generate a user interface template during the process modelling phase and to generate a user interface for executing process instances.

Figure 3.9a shows hierarchical conceptual model structure. Figure 3.9b illustrate a sample representation for the conceptual UI Model elements as user interface mock up. In an implementation like the proof of concept prototype the mock up, elements are replace by their real UI Widgets equivalents. In Figure 3.9 the single elements of the conceptual model



(a) UML Class Diagram of hierarchical conceptual User Interface Model



Bonds and stocks are both securities, but the major difference between the two is that (capital) stockholders have an equity stake in the company (i.e., they are owners), whereas bondholders have a creditor stake in the company (i.e., they are lenders). Another difference is that bonds usually have a defined term, or maturity, after which the bond is

ID	Period	Account	Value
1	Current Trend	2342	120.00 $
2	Last Week Trend	2356	123.20 $

(b) User Interface Model: Sample Screen Elements

Figure 3.9: User Interface Model to Screen Element Relations

are linked with the sample user interface elements by numbering IDs (1 - 9). Subsequently the single elements and their relations with each other are introduced.

The topmost element of the UI Model is called **UserInterface** (ID: 1), it is a container element for all subsequent elements. The UserInterface element includes another container elements called **FormTabTemplate** (ID: 2). FormTabTemplates are used to arrange all subsequent user interface form elements which are necessary to manipulate data based on a form. The accordion UI mock up element on the left side of Figure 3.9b is used to indicate the circumstance of multiple FormTabTemplates. Additionally it groups for a user form elements which logically belongs together. FormTabTemplate elements can contain multiple **ElementGroups** (ID: 3) which are used to arrange and group (complex) data input and output elements. For instance, a complex data element would result in an ElementGroup containing all its subsequent data elements. Labeled with the name of the complex data element as heading (right sided ID: 3). In addition, ElementGroups can be used to arrange data elements according to their associated overall semantic (left sided ID: 3). Similar as FormTabTemplates ElementGroups can be nested in each other for more advanced UI element arrangements. Due to the possibility of nesting elements, an instantiated UI Model becomes a tree like data structure.

A **FormElement** (cf. Figure 3.9a) is the abstract base element for all interaction elements contained in a FormTabTemplate (ID: 4 - 7 & 9). The two major representations of FormElement are **OutputElement** (ID: 5) and **InputElement** (ID: 6). InputElements are used for the input of new data and OutputElements to show existing data. A special case is the **EditElement** (ID: 7) this represents FormElements, which are a combination of both aspects from Input- and OutputElement. For example, the edit behaviour is needed if existing data is displayed and this existing data should be revised.

The last major aspect of the UI Model is the **ControlFlowNavigation** (ID: 8). This is a special FormElementGroup which is used for the explicit navigation between the single FormTapTemplates. Each FormTapTemplate contains exactly one ControlFlowNavigation element. The ControlFlowNavigation exist of exactly three **ControlFlowNavigationElements**. The most important of them is the **Next** ControlFlowNavigationElement ('**OK**' button in the sample screen of Figure 3.9b) (ID: 9). It is used for completing (send) a FormTabTemplate and thus all underlying data, including potential input data validations. The **Cancel** element just deletes all new input data of the actual FormTapTemplate. The **Previous** ControlFlowNavigationElement ('**«**' button in the sample screen of Figure 3.9b) can be used to get one step back, according to the set of FormTabTemplate elements contained in a UI Model.

For the present, these are enough general UI Model specifications to defined and understand the Transformation Patterns, which are introduced in Chapter 5.3 and Chapter 5.4. Specific details are discussed, in turn, during the presentation of single Transformation Patterns, as they are needed. Once again, it should be mentioned that the UI Model is a data structure that describes user interface elements and not the data structure for the UI elements itself. In other words, the UI Model stores the metadata that is necessary to create real user interface elements.

4 Requirements

In subsequent requirements for the generation of complex user interfaces based on block-oriented process models are defined. Starting with the presentation of sample use cases for complex user interfaces in Chapter 4.1. Followed by a definition of such complex user interface requirements in Chapter 4.2, which includes a discussion of user interface generation issues based on previously presented research work.

4.1 Use Case Process Models

In this chapter, three use cases will be introduced. Together with the related work, they build the based for the subsequent extraction of UI generation requirements. Use Case 1 in Chapter 4.1.1 describes a process for issue management. Use Case 2, presented in Chapter 4.1.2, handeles the configuration and ordering of a new car through an online configurator. Use Case 3 in Chapter 4.1.3 describes the creation of a new bank account.

For each of the presented use cases an initial description including the process models course of action is described. In addition, the special purposes according to UIs and their generation are outlined.

4.1.1 Use Case 1: Issue Management

The issue management process model shown in Figure 4.1 is based on the process of issue management systems, e.g., as they are used in the software development domain. In this simplified example, the processing of an issue contains of the following five activities:

1. In the **Open Issue** activity the issue is initial created an assigned to a user.
2. In the **Progress Issue** activity, the assigned user marks the issue as started and takes care of the issue.

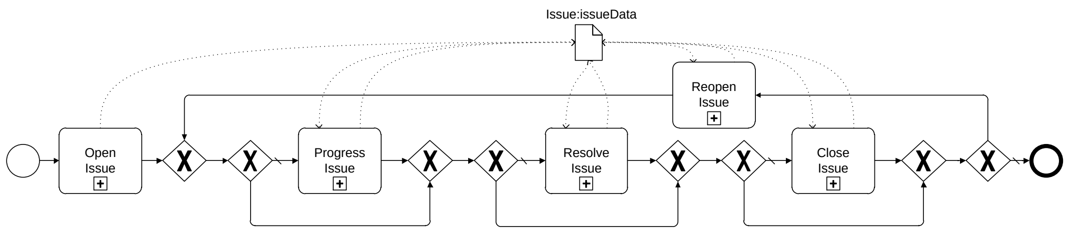


Figure 4.1: Process Model Example, Use Case 1: Issue Management

3. In the **Resolve Issue** activity the user checks whether the issue is resolved or not and marks it resolved which includes the writing of a necessary comment about the issues solution.

4. In the **Close Issue** activity the default end of an issue life cycle is reached, this results in a removal of the issue in the list of actual issues of a user.

5. The **Reopen Issue** activity is a optional step, e.g., if the solution was not adequate.

In this sample, process there exits two roles. The default role is the editor, which is allowed to execute the Open, Progress and Resolve activities. The administrator role has permission to execute all activities including Close and Reopen. The single activities of the process are all modelled as subprocesses. Each of the subprocess expect the Open activity the processed issue data is loaded from a storage system (service activity), edited by a user (human activity) and then stored again (service activity). The Open activity differs from this default subprocess since it does not contain the initial activity that loads the issue data. All subprocesses expect the open step are included in XOR control-flow blocks that include an empty execution path to bypass the activities since they are all optional.

The intent for this process model according to the user interface generation approach is to provide a comparatively simple example for initial testing. However, this process model addresses the feature of control-flow block detection and the distinction for role specific user interfaces.

4.1.2 Use Case 2: Car Configurator

The car configurator process use case is based on a web-based car configurator application like most of the car manufacturer provide on their web pages[1]. Figure 4.2 shows the simplified version of such a car configuration process a car dealership in which most of the single steps have been replace by subprocesses. A more detailed process model can be found in appendix A.2 (Figures A.1 and A.2). In the following, the single simplified activities of this model are described:

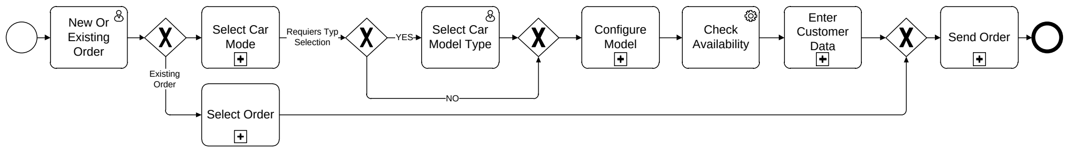


Figure 4.2: Process Model Example Overview, Use Case 2: Car Configurator

1. **New Or Existing Order** In this first activity the clerk has to decide if a new order item should be created or an existing one should be reloaded, e.g., by asking the customer.
2. **XOR Control-Flow Block** This block consists of the two following execution paths (options):
   a) **Create a new Order** This is the default execution path of the XOR control-flow block in which the following activities have to be executed to create a new car order:
      i. **Select Car Model** In this activity (subprocess) a list with all existing car models is loaded and presented to the clerk. The clerk has to select one of the car models
      ii. **Select Car Model Type** The Select Car Model Type activity is embedded in a XOR block since only for some of the car models different types are available.
      iii. **Configure Model** In this activity the car can be configured individually to the requirements of a customer. The activity can only be finished if the selected configuration is valid.

[1]An example for such an car configruator e.g. can be found at: `http://konfigurator.audi.de/`, last checked January 30, 2012

iv. **Check Availability** After the car has been configure in this activity the configuration system has to check the availability of the car and calculate a respective date of delivery.

v. **Enter Customer Data** The last activity of creating a new order is the entering of the customers personal data like his name, address, etc. In addition, a test drive date is appointed.

b) **Reload an Existing Order** This is the alternative execution path of the XOR control-flow block in which a existing order is reloaded.

i. **Select Order** In this subprocess the clerk selects a order item from a list of already existing orders. This is possible since the order item could have been created at an earlier time.

3. **Send Order** In this activity the clerk can decide between placing the order or storing the order for placing it at a later time. E.g., if the customer needs time to think about to buy the new car or not and if he or she has made the decision and comes back the already created order can just be reloaded from the system.

In this process model use case there exists only a single role, which is the clerk entering the data in the car configuration system as required by a customer. Hence this will results in the generation of a single UI Model instance. This car configuration process model use case consists of a more complex nesting-deep of control-flow blocks, which will in turn result in a more complex user interface then the previously outlined issue management example.

4.1.3 Use Case 3: Bank Account Creation

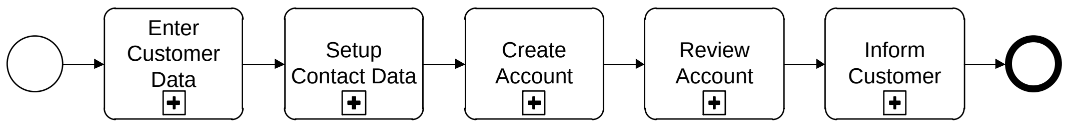


Figure 4.3: Process Model Example Overview: Bank Account Creation

Figure 4.3 illustrates an use case of the creation of a business banking account. Again, the single control-flow blocks haven been replace with subprocess elements as placeholder for simplicity reasons. The details of this process model can be found in the appendix A.2 (Figures A.3 and A.4). The basic concern behind this process is the creation of a new business account at a bank for a new or an existing customer. In the following some details for the course of action of the single steps as activities in Figure 4.3 are presented:

1. **Enter Customer Data** In this first step a bank clerk of the accounts division has to verify if the customer who wants to create an account is a new customer or an existing customer and create or edit the customer data afterwards.
2. **Setup Contact Data** In the next step the contact type according to the required communication for this new business account has to be specified. This might involve the update of existing contact data.
3. **Create Account** This step is determinate to specify all the required details for the account setup including the choice of the account type and customer specific adjustments.
4. **Review Account** As control mechanism this has to be performed by the head of the accounts division. It requires the verification of the business account data to create and a decision whether to accept or reject the account creation.
5. **Inform Customer** As a last step the clerk has to create a letter which informs the customer about the creation of his/her new account or inform him/her that the request was rejected.

This bank account creation process model combines the quirks of both previous use Cases. On the one hand, two role specific UI Model instances have to be created. On the other hand both UI Model instances and thus the resulting user interfaces are more or less complex according to the overall user interface elements to create and the nesting deep of these elements.

4.2 Complex User Interface Requirements

In the following, the previously presented use cases will be discussed according to their requirements for a user interfaces, which enables a user to effectively execute process instances. An efficient process execution is the *meta requirement* for other requirements. The use cases UI requirements are supplemented with aspects, which have been discussed in the related work chapter. The result of this discussion is a list of requirements for complex user interface generation.

For use case *Issue Management* each of the activities requires a similar looking user interface screen for processing the respective data elements. For each single primitive data element, it is important to distinguish between the different kinds of accessing data elements. For instance the system specific ID of an issue is a read only data field that is not allowed to be changed by a user, a generated user interface has to take care of this. This leads to another requirement according to this use case. There exist two different roles, thus role specific user interfaces are required. Since the user with the role editor is not allowed to execute the close and reopen activity.

The second use case *Car Configurator*, consists of a process model with more single activities and a more complex control-flow structure. For a user interface that enables an effective execution of such a car configuration it is necessary to summarize the processing of multiple activities in one user interface screen element. In addition, it is necessary to initialize the single user interface elements in conjunction with the decisions made by the user during the process executing. For instance, a user requires a total different user interface if an existing order element is loaded instead of creating a new order.

The process model for the *Bank Account Creation* use case consists of multiple control-flow block elements, which are connected in sequence with each other. Therefore an element in the user interface indicating the overall process execution state would result in a usability improvement. Loop elements, which result in executing activities for multiple times during process execution, have to be supported by the user interface. For instance, if data input made by a user is not valid and he or she has to enter the data values another time, a highlighting of the wrong data input including an error message is eligible. Another user interface related requirement to the third use case is the possibility of stepping back in the control-flow and retroactive adjusting a decision made before. For example, to change the contact data of the customer after the configuration of the account is already finished. To realize this it would be necessary to step back in the control-flow and re-execute the edit customer contact activity.

To realize the user interface requirements of the use cases, it is essential to keep the relation between single UI elements and process model elements. Sousa et al. have already proposed this in their work about user interface alignment with companies business processes [SMV10]. For the generation of role specific UIs a definition for different roles is required. This can be realized by linking the process activities to an organizational model, or more precisely to entities (e.g. a specific agent definition) of an organizational model [ZZHM07].

The introduced task models somehow cover the summarizing of single activities. Based on mapping the definition of a task as steps to reach a certain goal, multiple activities can be grouped and displayed in a single UI screen for their processing. The following list summarizes the requirements for the generation of complex user interfaces based on process models:

1. Distinction between different process model activities and different used data elements
2. Analyze of the process model control-flow structure
3. Traceable reference to an organizational model for role specific allocation of activities
4. Reference between process model and user interface elements and vice versa
5. Interaction options for retrospective activity modifications
6. Indication mechanism for the actual process execution state
7. Multiple UI instantiation options to consider conditional executed process model parts

With the help of this requirements, the User Interface Model and the process model definitions, in the next chapter the basis for an overall process model to user interface transformation is defined.

5 Transformation Patterns

With the help of the basic process model concepts, the User Interface Model and the cognition of the related work, including its outlined limitations, previously introduced, this chapter starts with an overview of the developed Transformation Model. This Transformation Model allows the transformation of block-oriented, activity-based process models to user interfaces. Therefore the model follows a top-down approach separated into four general steps (Chapter 5.1). This is followed by Chapter 5.2, which defines the concept of patterns according to our user interface generation approach and outlines a classification overview of the subsequent defined Transformation Patterns.

Chapter 5.3 and 5.4 consequently present the developed Transformation Patterns as the core part of the Transformation Model. Whereas **Elementary Transformation Patterns** (ETPs) in Chapter 5.3 handle basic process model transformations. The **Complex Transformation Patterns** (CTPs) in Chapter 5.4 then, in turn, link ETPs to realize the transformation of process control-flow blocks to user interfaces.

5.1 Transformation Model Compendium

The Transformation Model presented in this chapter enables the overall transformation of activity-based, block-oriented process models to user interfaces. This user interfaces enable end users to execute process instances of the respective process models. Figure 5.1 shows a UML class diagram of the hierarchical **Transformation Model** and its core parts. The diagram includes the interconnection, illustrated as UML compositions, between the single parts. In the following, four core parts of the Transformation Model are introduced, according to their numbering in Figure 5.1.

1. **Activity Transformation** covers the elementary aspect of generating user interface elements for each activity which needs user interaction. The specific details are described by the so called *Elementary Transformation Patterns* in Chapter 5.3.

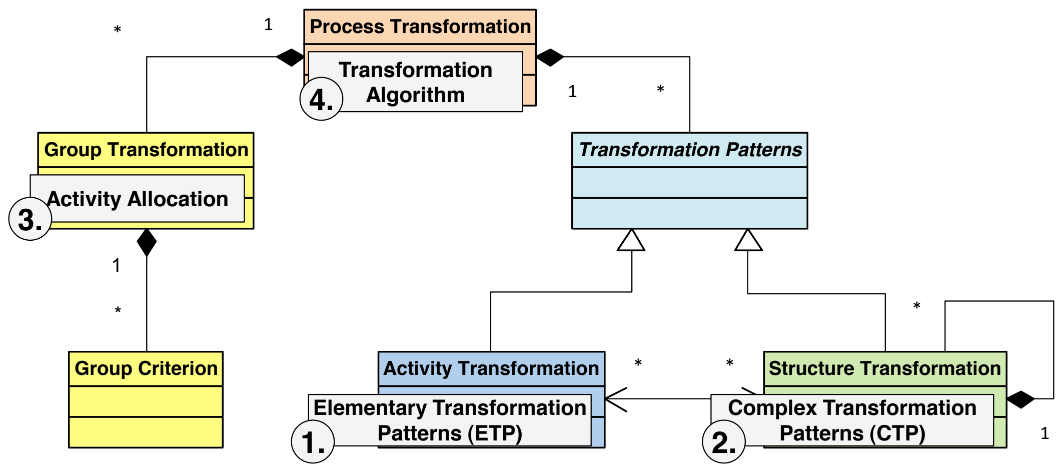


Figure 5.1: Transformation Model to generate User Interfaces for Activity-based Block-oriented Process Models

2. The **Structure Transformation** analyses the control-flow structure of a process model and extracts control-flow blocks. The extracted control-flow blocks are then, in turn, processed by the *Complex Transformation Patterns* which are introduced in Chapter 5.4. These CTPs apply the Activity Transformations described by the ETPs for each relevant activity included in the control-flow block, which they process. An important aspect for processing the control-flow blocks of a process model is the consideration of their nesting and their connection with each other.

3. The **Group Transformation** deals with the aspect that activities executed by human resources can be assigned to certain access groups. Access groups are often defined in an organizational model for human resources [LRD11]. In respect to this access groups the activities have to be allocated to different user interfaces since the execution of such a multiple role processes requires the interaction of different human resources. For each human resource role, a distinct user interface is needed. Due to the fact that the aspect of **Group Transformation** itself is substantial and rather complex and its interconnection with the overall process model transformation has to be defined, it is described in the Chapter 6.1.

4. The overall **Process Transformation** is the central component which links the Transformation Patterns and the Group Transformation. It interconnects the single parts in a well-defined **Transformation Algorithm** which is presented in Chapter 6.2.3.

Beside this short introduction of the Transformation Model core parts, in the following a classification overview of the Transformation Patterns is given. The classification is supplemented with some general prerequisite for the definition of patterns.

5.2 Overview of Transformation Patterns

Elementary Transformation Patterns (ETP)			
Category: Process Activity		**ETP 3**	**Data Transformation**
ETP 1	Human Activity to User Interface Transformation	*Category: Data Access*	
ETP 2	Non-Human Activity Transformation	**ETP 3.1**	Read Transformation
Category: Data Type		**ETP 3.2**	Write Transformation
ETP 3.4	Primitive Data Types	**ETP 3.3**	Read-Write Transformation
ETP 3.5	Complex Data Types (Business Objects)		

Complex Transformation Patterns (CTP)			
Category: Control-Flow Block Transformation			
CTP 1	**Sequence Block Trans.**	**CTP 4**	**Loop Block Trans.**
CTP 2	**Parallel Block Trans.**	**CTP 5**	**Subprocess Block Trans.**
CTP 3	**XOR Block Trans.**		
Category: Behaviour Block Transformation			
CTP 6	**Background Activity Trans.**		

Table 5.1: Process Model to User Interface Model Transformation Patterns Overview

As Table 5.1 shows, the major differentiation between the single Transformation Patterns is based on their operational level. The *Elementary Transformation Patterns* (ETPs) are applied on the level of a single process elements (e.g., activity, data elements). Whereas the *Complex Transformation Patterns* (CTPs) are used for the transformation of process control-flow blocks or certain activity groups. Since only a combined execution of both pattern types can lead to an overall end-user interface, the order of the pattern application is an important point. In general, the pattern application is based on a *top down approach*. Which means in the case of distinction between ETPs and CTPs, first to apply the CTPs to transform all detected blocks of a process, followed by the application of the ETPs to transform the single activities contained in this blocks.

The ETPs are split into three subcategories, the first subcategory is *Process Activity* which consists of two patterns to determine between activities which are executed by a human resource and those executed by the BPMS. The second subcategory is *Data Type* and the last is *Data Access* since the processed data elements of an activity are a key indicator for the Elementary Transformation Patterns.

CTPs are split into two subcategories. The base elements for all CTPs are control-flow blocks or certain parts of a process model. Thus, the first subcategory of CTPs is *Control-Flow Block Transformation*, which consists of patterns which are used to transform the most common control-flow blocks in process models, as defined in Chapter 3.1.1, to User Interfaces Model elements.

The subcategory *Behaviour Block Transformation* refers to a group of activities which represent a desired behaviour according to data processing and its relation to user interfaces. The patterns for this category are not completed yet. It is listed here since previous research has shown up that there are more parts or certain groups of activities in process models which need special handling according to the generation of complex user interfaces [Kam11]. More insides of the cause for this special handling are presented in the subsequent CTP related Chapter 5.4.2. The details of the single Transformation Patterns and of the introduced pattern categories are handled in the subsequent chapters starting with Elementary Transformation Patterns in the following.

5.3 Elementary Transformation Patterns

Elementary Transformation Patterns (ETP) are patterns, which describe basic process model to user interface transformations, e.g., how input fields for complex data elements of an activity are generated. The application level of ETPs is always a single activity. The source for the ETPs is a formalization of the state of the art way as actual BPMS, like presented in Chapter 2.2, generate user interfaces for single human activities. Referring to the requirements of the presented use cases, ETPs realize the requirement of differentiating between different activity types and analyze the used data elements. As outlined in Chapter 2.1.2 some of the aspects according to data in process models described by ETPs are more or less covered in previous work. In Künzle et al. the transformation of different (primitive) data types to respective form elements is introduced, but with a data object focused perspective of process models instead of activity based [KR11a]. The patterns according to

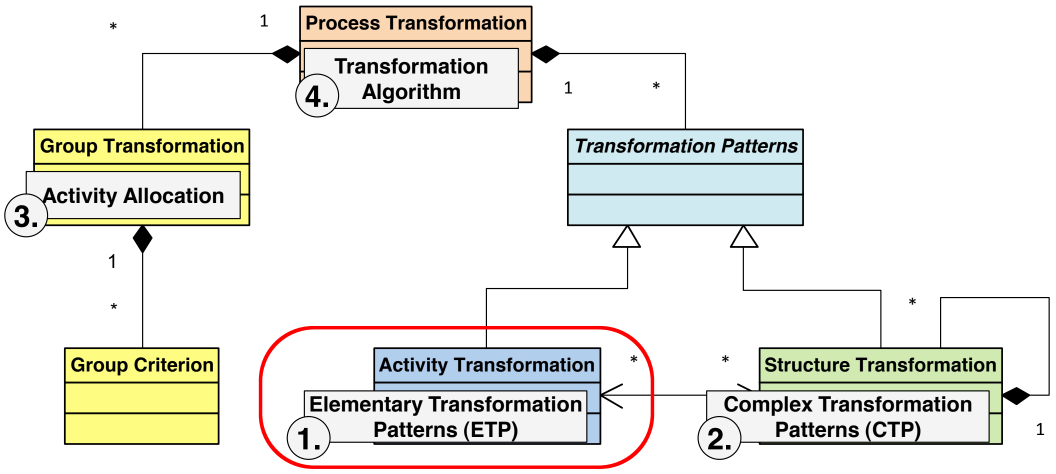


Figure 5.2: Elementary Transformation Patterns in the Transformation Model

data-flow and data visibility in process models, introduced by Russel et al. can be seen as a prerequisite for the here introduced Transformation Patterns [RtHEvdA05]. Since these data-flow patterns capture the more common aspects according to data in process models, like visibility of data, the kind of data interaction and the way data is transferred between single process components.

ETPs are used by CTPs to transform the single activities of more complex control-flow blocks or groups of activities to distinct form elements described by the User Interface Model. In addition, the application order of single ETPs is relevant to realize an overall process model to end-user interface transformation. Thus the ETPs described in the following chapters are ordered according to their execution in the overall process model transformation (c.f. Chapter 5.1). Starting with *Human Resource Process Activities* in Chapter 5.3.1 followed by patterns which are responsible for the correct transformation of an activity's data flow in Chapter 5.3.2.

5.3.1 Human Resource Process Activities

The definition for the Transformation Patterns used in the following has been aggregated by reviewing the several works about patterns according to process models and BPM in general [WRR07, vdAtHKB03, RtHEvdA05, LRtHW$^+$11, LRWM$^+$11]. Table 5.2 holds the definition for the first of the Transformation Patterns, *ETP1*. It is defined by a unique **Pattern Name** including an acronym for later reference as heading of the table. The **Description** gives a short, general textual explanation of the pattern. With the help of an **Example** the pattern is linked to a real life problem. A **Figure** illustrates the required actions to transform a certain process model element (specified using BPMN) to a user interface. The user interface is illustrated as screen mock-up. The **Problem** description refers to the situation in which the application of the pattern is necessary. The **Prerequisites** define required prerequisites for a process model to enable the implementation and application of the pattern. The **Implementation** property defines how the pattern can be implemented with reference to the Prerequisites and to the instantiation of required UI Model elements (c.f. Chapter 3.2.2). The **Related Patterns** definition references related patterns.

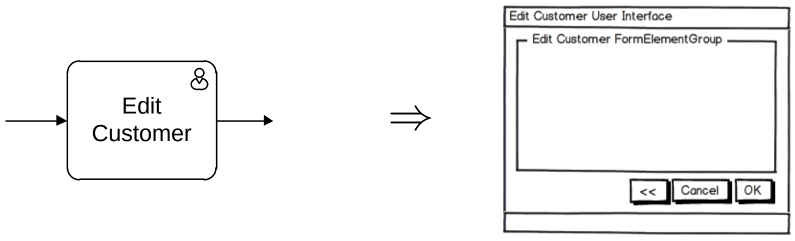
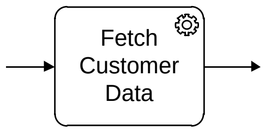
ETP1: Human Activity to User Interface Transformation	
Description:	Human Activities (i.e., activities which need a human resource for their execution) with input and/or outputdata elements need a (G)UI-based form for their execution.
Example:	A bank clerk has to edit the data of a customer.
 Edit Customer ⇒ Edit Customer User Interface Edit Customer FormElementGroup << Cancel OK	
Problem:	Human Activities need a User Interface.
Prerequisites:	Data about the type of Activities is required.
Implementation:	All activities of the type human (need a human resource for their execution) need to be considered for user interface generation. Referring to the UI Model elements for each activity a FormElementGroup has to be generated.
Related Patterns:	ETP2

Table 5.2: ETP1: Human Activity to User Interface Transformation

An elementary precondition for the user interface generation based on process models is the recognition of activities, which need a human resource for their execution. Thus, the distinction of those human resource activities between activities which do not need a human resources for their execution is necessary. The patterns of the category *Human Resource Process Activities* introduced in this chapter handle this aspects. *ETP1* describes the basics required for the recognition and transformation of human resource activities. The details for the allocation of human resources activities to distinct User Interface Model elements are a complex topic. It is addressed by the Complex Transformation Patterns in Chapter 5.4 and in subsequent Chapter 6 with covers the various details of activity allocation. Therefore, *ETP1* listed in Table 5.2 only describes the fact that it is necessary to generate *User Interface* elements based on the UI Model introduced in Chapter 3.2.2 for the processing of such a human resource activity. In addition, it is necessary to interconnect the newly create UI Model elements with existing ones, e.g., the parent UI Model elements generated for the control-flow block this activity belongs to.

ETP2: Non-Human Activity Transformation	
Description:	Non-Human Activities are only indirect relevant for UI generation. They have to be considered for the generation of source code.
Example:	A customer data set has to be fetched from an external CRM system.
	$\Rightarrow$ `public class AccountCreationProcess{ public Customer fetch(Integer customerID){ // fetch customer data from CRM system Customer customerData = ... return customerData; } }`
Problem:	To generate a user interface that is capable of executing process instances it is necessary to generate source code for the non-human (service) activities and their interactions between the human activities.
Prerequisite:	Data about the type of Activities and their data- and control-flow are required.
Implementation:	For all non-human (service) activities, source code stubs have to be generated which serve as interconnecting code for human activities.
Related Patterns:	ETP1

```
public class AccountCreationProcess{
  public Customer fetch(Integer customerID){
    // fetch customer data from CRM system
    Customer customerData = ...
    return customerData;
  }
}
```

Table 5.3: ETP2: Non-Human Activity Transformation

The counterpart to *ETP1* is the pattern *ETP2: Non-Human Activity Transformation*. As *ETP1* it builds upon the criteria to differ between human- and non-human activities. Listed in Table 5.3 the pattern describes the fact that it is necessary not just to ignore activities, which do not need direct user interaction. Since the overall process model to end-user interface generation should result in a set of user interfaces which are capable to execute process instances, it is even essential to generated code stubs for this *non-human activities*. This is caused by the fact that the process data-flow can include all kind of activities and if the non-human or service ones would be ignored a suitable data element processing would be impossible. A simple example for such a situation would be a non-human activity, which fetches data from an external system, to display this data to a user in a subsequent human activity. If the non-human activity was just ignored the result would be that there is nothing to display to the user. The more specific details about the integration of non-human activities in the user interface transformation process are covered in Chapter 6.2.3. The following chapters presents all Elementary Transformation Patterns, which are relevant according to data transformation specifics.

5.3.2 Data Transformation

ETP3: Data Transformation listed in Table 5.4 is the base Transformation Pattern for the data transformation category of all ETPs. It describes the fact that UI Model FormElements are generated based on the interaction of process model activities with data elements.

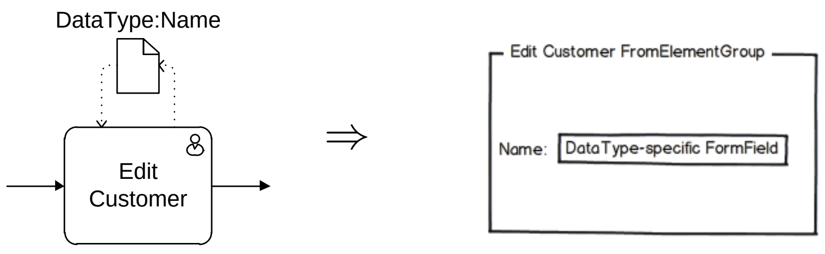
ETP3: Data Transformation	
Description:	If human activities use data elements, the generated user interfaces need FormElements (that include a prefix label) which have to be generated according to the data types of the data elements.
Example:	In an activity, the data of a customer is edited.
 DataType:Name Edit Customer ⇒ Edit Customer FromElementGroup Name: DataType-specific FormField	
Problem:	Human activities, which interact with data elements, need forms with fields and labels for these data elements.
Prerequisite:	Detection of relevant data elements based on the activities data input and output elements.
Implementation:	The FormElementGroups generated by ETP1 have to be filled with FormElements according to the data elements used by an activity.
Related Patterns:	ETP1, ETP2

Table 5.4: ETP3: Data Transformation

The following three patterns build upon *ETP3* and deal with the differences between data element access of activities. Thus for *ETP3.1*, *ETP3.2* and *ETP3.3* the *type of data element* *access* by an activity is a shared prerequisite. The first of this three data transformation patterns handles the FormElement generation for data elements that are accessed in a writing manner by activities in a process model. Thus, it describes that FormElements with the possibility of data input have to be generated. Table 5.5 holds the details for this pattern called *ETP3.1: WRITE Data Transformation.*

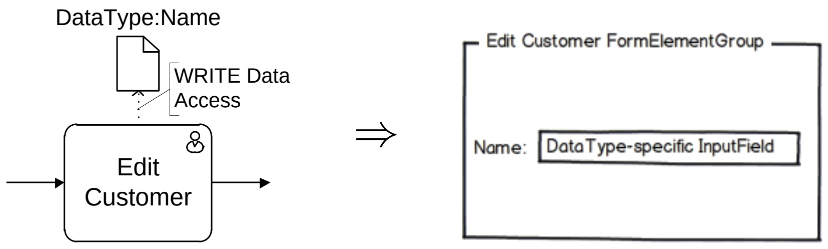
ETP3.1: WRITE Data Transformation	
Description:	A WRITE data element requires an input field inside the user interface form of the respective activity. The value entered in the UI Widget associated with an UI Model InputField is used as value for the WRITE data element.
Example:	In an activity, the new data values for a customer are entered by an agent in a customer edit form.
 DataType:Name WRITE Data Access Edit Customer ⇒ Edit Customer FormElementGroup Name: DataType-specific InputField	
Problem:	To write data elements for an activity a user interface with the possibility of interaction is needed.
Prerequisite:	Data elements have to be handled different, according to the way they are accessed (read, written or read & written) by an activity.
Implementation:	During the FormElement generation UI Model InputFields have to be generated according to the used WRITE data elements of an activity.
Related Patterns:	ETP1, ETP2, ETP3

Table 5.5: ETP3.1: WRITE Data Transformation

As a counterpart to *ETP3.1*, *ETP3.2* which is the second of the data Transformation Patterns handles the FormElement generation for data elements which are accessed in a reading manner by activities in a process model. Therefore, it defines the generation of UI Model OutputElements. Table 5.6 lists the details for this pattern called *ETP3.2: READ Data Transformation.*

The last of the three data Transformation Patterns is *EPT3.3: READ-WRITE Data Transformation.* Table 5.7 shows the details for *ETP3.3.* The pattern is a combination of *ETP3.1*

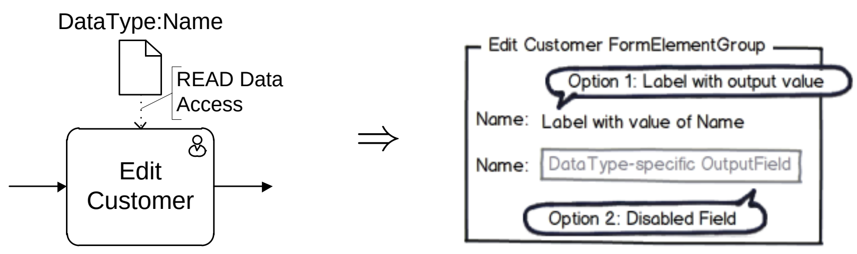
ETP3.2: READ Data Transformation	
Description:	A READ (only) data element creates a label with the value of the data element inside the element group of the respective activity.
Example:	During editing of customer data, some of the data elements, e.g. a customer ID, are immutable. Thus, these elements are only displayed to an agent processing the respective activity.
 DataType:Name READ Data Access Edit Customer ⇒ Edit Customer FormElementGroup Option 1: Label with output value Name: Label with value of Name Name: DataType-specific OutputField Option 2: Disabled Field	
Problem:	For Read data elements, the possibility to display their values to a user is required.
Prerequisite:	Data elements have to be handled different, according to the way they are accessed (read, written or read & written) by an activity.
Implementation:	During the processing of READ data elements of an activity UI Model OutputElements are generated and allocated to the respective FormElementGroup. As shown in the example OutputElements have two different options for their UI Widget implementation (disabled input field & text label).
Related Patterns:	ETP1, ETP2, ETP3

Table 5.6: ETP3.2: READ Data Transformation

and *ETP3.2*. It specifies the situation in which a data element is access in both, read and write (edit), manner. The notation as additional pattern is necessary since a simple application of *ETP3.2* followed by application of *ETP3.3* would result in two separate FormElements. One form InputElement for writing the data element and one form OutputElement to display the read value of the data element. However, this is not the desired behaviour. Only a single edit element should be generated. This UI Model EditElement is, in turn, filled with the value of the read data element. The value of the input form can be change (edited) by a user and is used to write the data element during the processing of the activity, e.g., if sending the respective FormTabTemplate by pressing the OK button.

The now following two Transformation Patterns handle the differences according to the data types of the data elements used by activities. Therefore pattern *ETP3.4* and *ETP3.5*

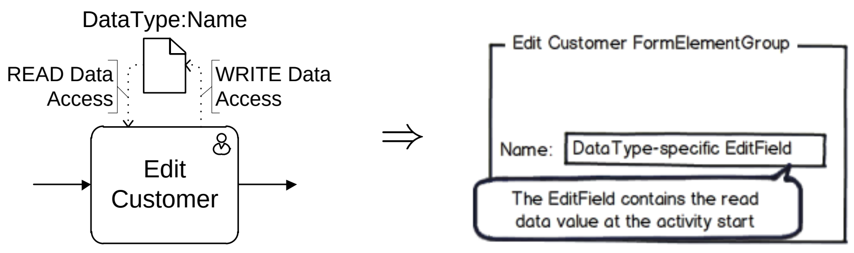
ETP3.3: READ-WRITE Data Transformation	
Description:	A READ-WRITE (edit) data element creates an input field with the value of the data element inside the element group (form) of the activity. The value in the input field can be changed (edited). After sending the respective form the value of the input field is used as new value for the data element. This pattern is a combination of ETP3.1 & ETP3.2.
Example:	In a customer edit form, already existing customer data values can be changed.
 DataType:Name READ Data Access WRITE Data Access Edit Customer ⇒ Edit Customer FormElementGroup Name: DataType-specific EditField The EditField contains the read data value at the activity start	
Problem:	To read & write (edit) the same data element a user interface with the possibility of interaction & to display values is needed.
Prerequisite:	Data elements have to be handled differently, according to the way they are accessed (read, written or read & written) by an activity.
Implementation:	During the processing of an activity's data elements that are accessed in READ & WRITE manner UI Model EditElements have to be generated. If the respective activity is started, the EditElements have to be initialized with read data value. If the respective activity is finished the updated value has to be written to the EditElement. The UI Widget representation for an EditElement is an input form field.
Related Patterns:	ETP1, ETP2, ETP3, ETP3.1, ETP3.2

Table 5.7: ETP3.3: READ-WRITE Data Transformation

share the *data type* as a common criteria. The first of them *ETP3.4: Primitive Data Type Transformation* listed in Table 5.8 describes the FormElement generation for a limited set of primitive data types. The data types *Integer*, *Float*, *Boolean*, *String*, *Date* and *URI* where selected since they have been identified as the most important primitive data types used in BPM systems and therefore in process models [RtHEvdA05]. The details for this identification can be found in Russel et al. a previously mentioned work about so called *Workflow Data Patterns* [RtHEvdA05].

Pattern *ETP3.4* describes a (set) of FormElements for each of this primitive data types which are generated during the activity to user interface transformation process. *ETP3.4* is related to the patterns *EPT3.1*, *EPT3.2* and *EPT3.3*, since the type of data access by an activity directly affects the kind of generated form elements. E.g., if a *String* data type is accessed in a reading manner (cf. *ETP3.2* in Table 5.6) a label holding the String value is generated. In contrast an input field is generated if the *String* data type is accessed in a writing manner (cf. *ETP3.1* in Table 5.5).

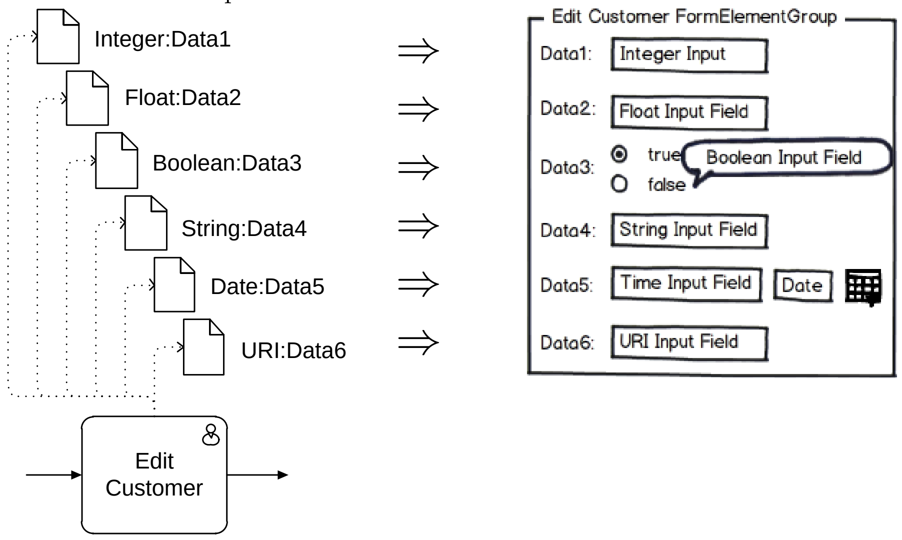
ETP3.4: Primitive Data Type Transformation	
Description:	Different primitive data types require the generation of different user interface form elements.
Example:	A form for editing customer data requires the processing of different primitive data types thus different interactions elements are required.  Integer:Data1 Float:Data2 Boolean:Data3 String:Data4 Date:Data5 URI:Data6 Edit Customer Edit Customer FormElementGroup Data1: Integer Input Data2: Float Input Field Data3: true false Boolean Input Field Data4: String Input Field Data5: Time Input Field Date Data6: URI Input Field
Problem:	For different primitive data types of data elements used by activities, different form elements have to be generated.
Prerequisite:	The characteristic (Integer, Float, Boolean, String, Date) of the primitive data type of the data elements used by an activity.
Implementation:	For the generation of applicable UI Widgets the created UI Model FormElements have to take the data type of an activities data elements into account.
Related Patterns:	ETP1, ETP3

Table 5.8: ETP3.4: Primitive Data Type Transformation

The last of the ETPs is *ETP3.5: Business Object Data Type Transformation.* It covers the fact that data types used in process models are often complex types represented as so called *business objects* [HYZL10]. These business objects or complex data types are composed of primitive data types and other business objects nested in the original business object.

Table 5.9 illustrates how to handle activities using such nested data objects according to UI Model FormElement generation for user interfaces. The general approach at this is to create FormElementGroups for the grouping of the data elements a business object consist of. An important supplementation to the transformation of business objects or more general

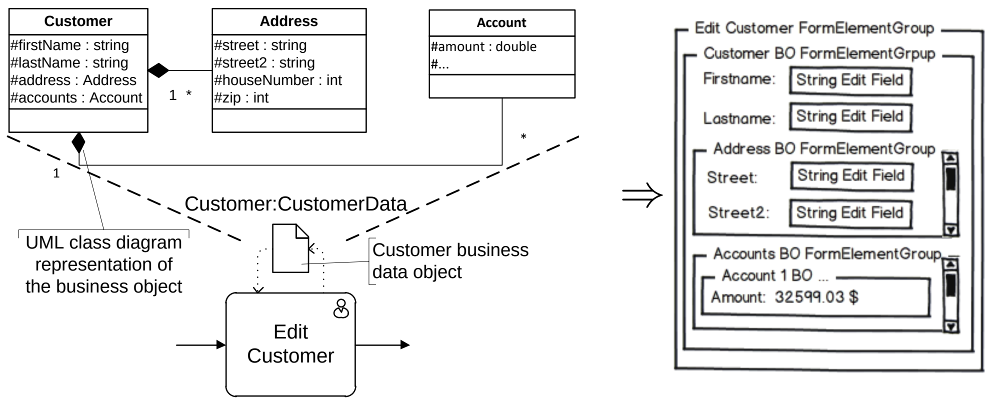
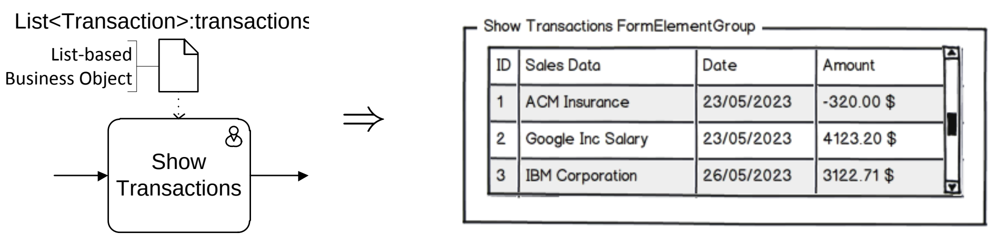
ETP3.5: Business Object Data Type Transformation	
Description:	For each business object, the primitive data types of which the business object consists have to be extracted. If a business object contains other nested business objects, the primitive data element extraction has to be done for each of this nested business objects recursively. For the extracted primitive data element form elements have to be generated like described in ETP3.4.
Example:	A *Customer* business object instance should be displayed in a form for editing customer data. This *Customer* business object can consists of several different business objects, e.g., the address of the customer, the account the customer is associated with, etc.
 Customer #firstName : string #lastName : string #address : Address #accounts : Account Address #street : string #street2 : string #houseNumber : int #zip : int Account #amount : double #... 1 * 1 * Customer:CustomerData UML class diagram representation of the business object Customer business data object Edit Customer ⇒ Edit Customer FormElementGroup Customer BO FormElementGrpup Firstname: String Edit Field Lastname: String Edit Field Address BO FormElementGroup Street: String Edit Field Street2: String Edit Field Accounts BO FormElementGroup Account 1 BO ... Amount: 32599.03 $	
Problem:	Business objects contain multiple primitive data elements which have to be extracted to use them to generate form elements.
Prerequisite:	The characteristic (complex, primitive) of data elements.
Implementation:	Before generating FormElements all primitive data elements of a business object have to be extracted. FormElements are grouped according to their business object membership.
Related Patterns:	ETP3, ETP3.4

Table 5.9: ETP3.5: Business Object Data Type Transformation

complex data objects is the handling of list based data types. The term list based data types refers to all kind of data types which are based on multiple instances of another data type. These kinds of data types require list-oriented or table-oriented UI Widgets for their display with an option to edit the data if this is necessary according to the respective process model data-flow. In the sense of the UI Model no additional elements are required since the single entities of each entry line are still FormElements. The allocation of single UI Model FormElements to a table structure is specified by an additional property of the FormElement. Figure 5.3 shows an example for the resulting UI Widgets for the transformation of a list of transaction business objects.


List<Transaction>:transactions
List-based Business Object
Show Transactions
⇒
Show Transactions FormElementGroup

ID	Sales Data	Date	Amount
1	ACM Insurance	23/05/2023	-320.00 $
2	Google Inc Salary	23/05/2023	4123.20 $
3	IBM Corporation	26/05/2023	3122.71 $

Figure 5.3: UI Widget Generation for a list of Transactions, Transformation of List-based Business Objects

Based on their limitations according to a single activity it is obvious that ETPs are not sufficient enough for the overall generation of complex user interfaces to handle complete processes instances. The interconnection, especially according to the process model control-flow, of activities has to be taken into account. The *Complex Transformation Patterns* introduced in the next address this facet.

5.4 Complex Transformation Patterns

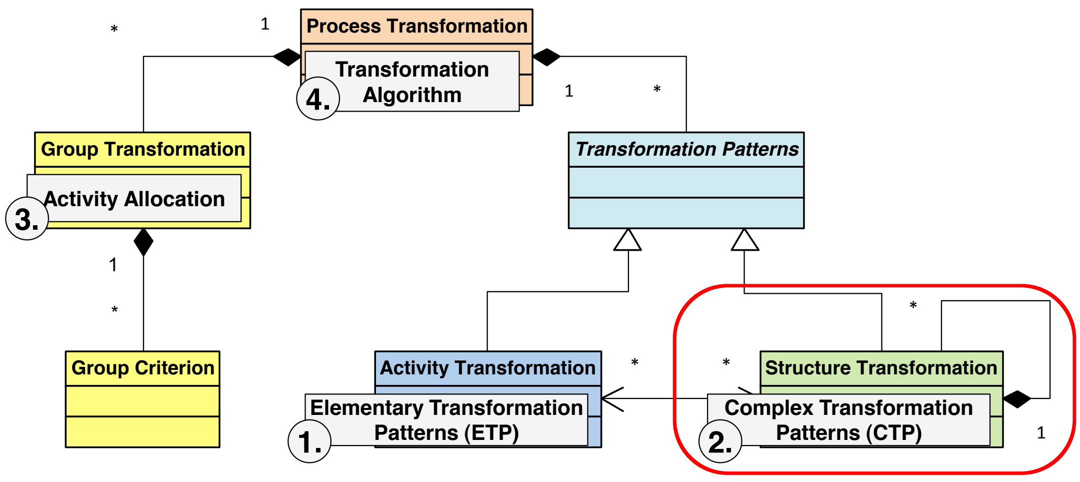


Figure 5.4: Complex Transformation Patterns in the Transformation Model

Complex Transformation Patterns (CTP) realize the complex control-flow structure transformation of a process model. They generate UI Model FormTabTemplate elements and interconnect them with each other as required according to process model control-flow structure. In addition, they use the ETPs introduced in the previous chapter for the transformation of the single activities included in the control-flow blocks. Process models which are addressed by these CTPs have to be block-structured like defined in Chapter 3.1.1.

The complex user interfaces respectively the UI Model elements, as they are generated by these control-flow block CTPs are to a certain extent obvious. Nevertheless they also follow user interface modelling principles, e.g., as defined by Paternò in the context of task-based UI modelling [Pat00] and hierarchical ConcurTaskTrees notation [PMM97]. ConcurTaskTrees notation has been introduced in Chapter 2.1.1.

These control-flow block CTPs are supplemented with so called *Behaviour Block* CTPs. The Behaviour Block CTPs are a perspective CTP category which seem to be reasonable according to previous conducted research work [Kam11]. The try to handle the semantics of certain process model snippets according to particular user interface generation issues. Referring to the complex user interface requirements (cf. Chapter 4.2) the CTPs realize the requirement of analyzing the control-flow block structure to transform this structure to a user interface. Additionally they address the aspect of realizing multiple instantiation options for a user interface. Similar to the application of ETPs a definition for the application of the CTPs is required. This definition is a important part of the overall Transformation Algorithm for user interface generation and is handled in Chapter 6.2.2.

5.4.1 Control-Flow Block Transformation

The first of the Complex Transformation Patterns (CTPs) is *CTP1: Sequence Block Transformation* which describes the aspects of transforming an arbitrary number of activities or detected blocks connected with each other by the process control-flow. Table 5.10 specifies the details for this pattern. Based on our process model definition, activities and control-flow blocks can be handled in the same manner. For each of them a UI Model FormTabTemplate element is generated and connected with the parent UI Model element according to the process model control-flow block nesting structure.

An additional point to mention according to the transformation of sequence blocks is based on the general conditions of our process model limitations (cf. Chapter 3.1.1). Consequential

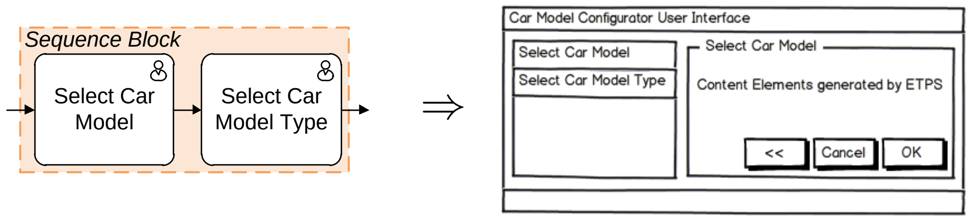
CTP1: Sequence Block Transformation	
Description:	Activities or more precise control-flow blocks, which are connected in sequence should result in a distinct user interface screen.
Example:	During the configuration of a car an agent has to select a car model in a first activity followed by an activity in which the type of the car model is selected.
 Sequence Block Select Car Model Select Car Model Type ⇒ Car Model Configurator User Interface Select Car Model Select Car Model Type Select Car Model Content Elements generated by ETPS << Cancel OK	
Problem:	For each activity or control-flow block contained in a sequence block a distinct user interface screen for the required interactions is necessary. If one form in a screen is completed the next form screen according to the process control-flow has to follow.
Prerequisite:	Sequence conrtol-flow structure of activities or control-flow blocks.
Implementation:	For each activity or control-flow block contained in a sequence block a UI Model FormTabTemplate element, including a ControlFlowNavigation element to link to precursor and successor, is generated and arranged in sequence.
Related Patterns:	ETP1, ETP2, ETP3, CTP2

Table 5.10: CTP1: Sequence Transformation

a process model consist of one basic sequence control-flow block at the topmost hierarchy level. Thus, the resulting UI Model always consists of a set of FormTabTemplate elements arranged in sequence at its topmost hierarchy level. At which an extreme would be a FormTabTemplate set with only a single element, e.g., if the whole process model consists of a single global XOR control-flow block. On a first look, the sequence transformation seems to be obvious. However, previous research [Kam11] has shown that it is useful to combine the processing of activities arranged in a sequence in a single user interface form. This controversy is handled as Sequence Problem in Chapter 6.1.2 in which a possible solution could be a manual marking of activities to achieve their processing in a single form.

Table 5.11 introduces CTP2, which transforms parallel process model control-flow blocks. The results of the application of this pattern are similar as the processing of a single activity. However, the pattern covers more than the transformation of two parallel-arranged activities. It also includes the aspect of processing arbitrary control-flow blocks parallel arranged to user

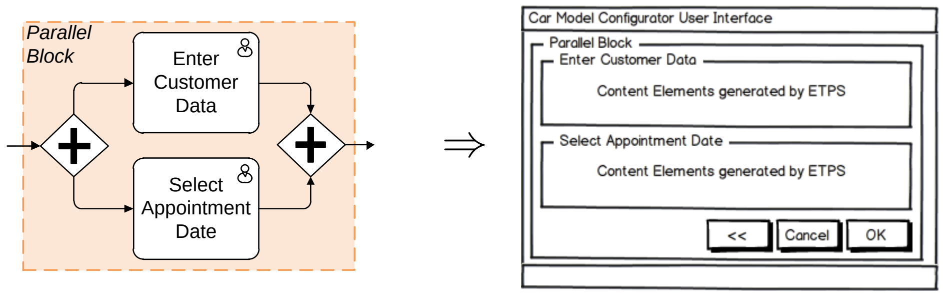
CTP2: Parallel Block Transformation	
Description:	Activities or nested control-flow blocks which are inside a parallel block can be summarized in a single user interface form for their processing.
Example:	While entering the data of a customer the selection for an appointment date can be performed in parallel.
	 Parallel Block Enter Customer Data Select Appointment Date ⇒ Car Model Configurator User Interface Parallel Block Enter Customer Data Content Elements generated by ETPS Select Appointment Date Content Elements generated by ETPS << Cancel OK
Problem:	Activities of a process model, which are in a parallel control-flow block, should be performed parallel when a single user or a specific user group performs it.
Prerequisites:	Parallel control-flow structure of activities or control-flow blocks, noticeable by parallel split and join gateways.
Implementation:	During form generation, parallel control-flow blocks have to be detected for each branch (execution path) of this blocks UI Model FormElementGroups associated with a FormTabTemplate, which represents the complete block in the UI Model have to be generated.
Related Patterns:	ETP1, ETP2, ETP3, CTP1

Table 5.11: CTP2: Parallel Block Transformation

interface elements. For instance, the replacement of one of the parallel-arranged activities by a complete sequence control-flow block would result in always displaying the form elements of the single activity in combination with each of the form elements from the sequence activities.

Additionally, the pattern is required for the traceable covering of all associations between process model and user interface elements as defined by the complex user interface requirements (cf. Chapter 4.2). In the sense of this traceable association, covering an important aspect is the consideration of process control-flow inside single user interface forms. Without that, it would be impossible to track the relation between interaction elements like input fields (represented by UI Model FormElements and its derivates) and the according data elements as well as activities of a process model. With the help of this so called micro level process modelling more advanced user interfaces can be generated [KR11a]. E.g., with the possibility to specify the order in which data should be entered in a form by splitting a single activity in multiple activities arranged in parallel or sequence with each other. The details for the additional topic of user interface to process model element linking is rather complex and thus covered by Chapter 6.2.4.

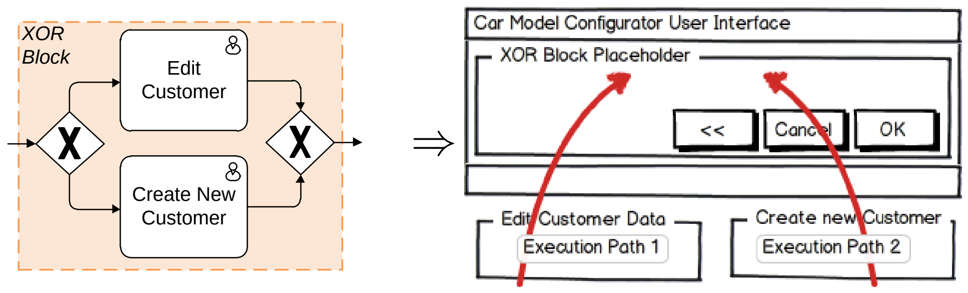
CTP3: XOR Block Transformation	
Description:	Activities or nested blocks which are inside a XOR (exclusive decision) control-flow block require multiple different user interface form elements based on the number of possible execution paths. The decision for which execution path the widgets are drawn inside a complex user interface is only possible during process run-time.
Example:	If a customer already exists, the edit customer form is displayed. If not a create new customer form has to be displayed instead.
 XOR Block Edit Customer Create New Customer X X ⇒ Car Model Configurator User Interface XOR Block Placeholder << Cancel OK Edit Customer Data Execution Path 1 Create new Customer Execution Path 2	
Problem:	Conditional executed paths of an XOR control-flow block in a process model require the generation of all UI Model elements for each of this paths during modelling time. While during run-time, only the elements of the chosen execution path are instantiated.
Prerequisite:	XOR control-flow blocks, noticeable by XOR split and join gateways. During run-time additionally the decision element which triggers execution path selection of the XOR split gateway.
Implementation:	A placeholder UI Model FormTabTemplate that holds all execution paths, which in turn link to all UI Model elements in this execution path, has to be generated for each XOR control-flow block. During run-time, only the elements of the chosen execution path are displayed inside of the placeholder element.
Related Patterns:	ETP1, ETP2, ETP3, CTP2

Table 5.12: CTP3: XOR Block Transformation

CTP3: XOR Block Transformation, described in Table 5.12, is the first of the control-flow block patterns for which a direct and adequate visualization with the help of a single user interface mock-up is impossible. Since, it requires the generation of multiple distinct independent UI Model elements. To be more precisely, the generation of all UI Model elements, which are required for each execution path, the XOR control-flow block consists of. The decision for which execution path and its corresponding UI Model elements user interface widgets have to be drawn is impossible if there is only a process model. For this decision, run-time data of a corresponding process instance is necessary. Thus, it is only possible to generate real UI widgets during process run-time based on the process instance data. In particular, the required data for the decision which of the execution paths of the XOR control-flow block to execute is required.

This results in a limitation for the creation of complex user interfaces for process models, which include XOR blocks during the process modelling phase. To deal with this, also in respect to a visual complex user interface editor, a placeholder FormTabTemplate element that holds all possible execution paths and their referenced UI Model elements is generated for the XOR control-flow block. Figure 5.5 shows the resulting UI Model data structure for the XOR control-flow block sample of CTP3 (cf. Table 5.12) which includes the respective UI Model elements for both of the execution paths.

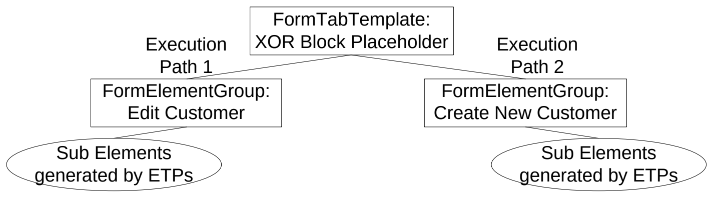

FormTabTemplate:
XOR Block Placeholder
Execution Path 1
Execution Path 2
FormElementGroup: Edit Customer
FormElementGroup: Create New Customer
Sub Elements generated by ETPs
Sub Elements generated by ETPs

Figure 5.5: Resulting UI Model Data Structure for XOR Block of CTP3 Example

The last important thing about the XOR block transformation again refers to the run-time of process instances. During this, a decision data element that triggers the XOR gateway and thus defines which execution path is executed is required. By default, the decision data element is send by an activity, which is previous of the XOR gateway in the process model control-flow. However, it is also possible that external circumstances trigger a XOR based decision, e.g., the time of day could be such an external circumstances, which is completely independent from the process instance data.

The details of *CTP4: Loop Block Transformation* Complex Transformation Pattern are shown in Table 5.13. Similar to previously introduced CTP3 it is not directly possible to create a UI mock-up to visualize the specific UI Model related implementation details for it. Based on our process model definitions from in Chapter 3.1.1, it is possible to nest arbitrary control-flow blocks and of course activities inside of loop blocks. CTP4 now defines that for each loop block a UI Model FormTabTemplate has to be generated and all process model

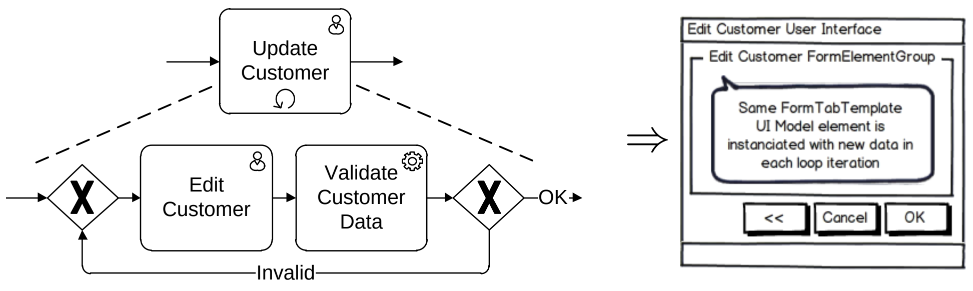
CTP4: Loop Block Transformation	
Description:	Activities or nested control-flow blocks, which are inside a loop block require additional UI Model actions according to their execution semantic.
Example:	The data processed by an activity for editing a customer data set is validated by a validation service activity. The corresponding edit customer data activity is executed as long as the performing agent has entered invalid data. In each iteration, a specific error message is generated and displayed to the agent.
 Update Customer Edit Customer Validate Customer Data OK Invalid ⇒ Edit Customer User Interface Edit Customer FormElementGroup Same FormTabTemplate UI Model element is instanciated with new data in each loop iteration << Cancel OK	
Problem:	Loop control-flow blocks can consist of multiple nested activities and control-flow blocks. In each loop iteration, the UI Model elements and their respective UI widgets in these nested activities have to be instantiated with correct data.
Prerequisite:	Loop control-flow blocks, noticeable by Loop start and end gateways. During process runtime the decision element which decides about quitting the loop.
Implementation:	All child UI Model elements of a loop block have to be associated with a FormTabTemplate element. In each iteration during process run-time, the value of the decision element for leaving the loop has to be checked.
Related Patterns:	ETP1, ETP2, ETP3, CTP1, CTP2, CTP3

Table 5.13: CTP4: Loop Block Transformation

elements nested in this loop block are associated with it. If the loop block contains more than a single human activity this results in a nested sequence control-flow block inside the loop block by default. Another similarity with CTP3 is the requirement of a decision element whereupon in CTP4 this decision element is required to decide about quitting the loop. The loop decision element has to be generated based upon the execution and the respective data processing inside the loop control-flow block during the execution of a process instance. Summing up CTP4 can be characterized as a marker pattern since it defines that all elements nested in a loop control-flow block have to be repeated based upon the interpretation of the value of a decision element.

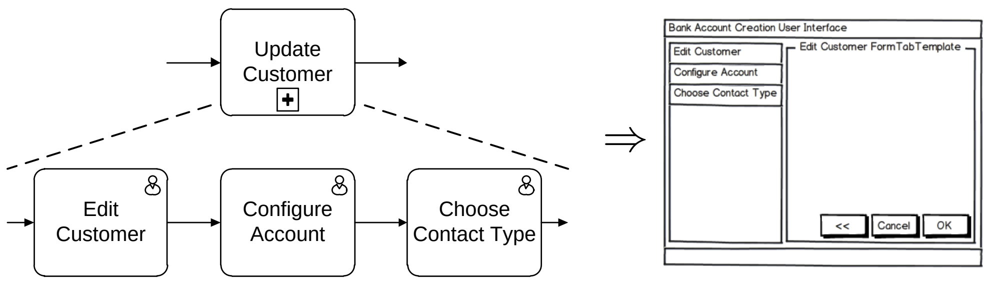
CTP5: Subprocess Block Transformation	
Description:	The elements of an underlying process model of a subprocess should be integrated in the parent process model for complex user interface generation.
Example:	A subprocess for updating a customer, according to an overall bank account creation process, consists of three sequential activities. Edit the customer data, followed by configure the account options and finally choose a contact type for the customer. The user interface widgets required for this three activities should be integrated in the global complex UI of the parent process.
	 Update Customer Edit Customer Configure Account Choose Contact Type ⇒ Bank Account Creation User Interface Edit Customer Configure Account Choose Contact Type Edit Customer FormTabTemplate << Cancel OK
Problem:	If a process model contains subprocess blocks the underlying process models for this subprocesses (respectively the process model elements they consist of) have to be integrated into the parent process model for an overall complex user interface generation.
Prerequisite:	Detection of subprocess blocks, resulting overall process model control-flow blocks nesting deep.
Implementation:	This pattern has to be executed before all other CTPs since it extracts the process model elements of the subprocesses to the global process model. In addition, a UI Model FormTabTemplate should be generated as parent element for all UI Model elements generated for the subprocess block.
Related Patterns:	ETP1, ETP2, ETP3, CTP1, CTP2, CTP3, CTP4

Table 5.14: CTP5: Subprocess Block Transformation

The details about pattern *CTP5: Subprocess Block Transformation* are defined in Table 5.14. It differs from all previously introduced patterns because it can be seen as a meta pattern. In this case, the term meta refers to the aspect that it is necessary to perform CTP5 before all other CTPs. Since it describes the requirement to extract the underlying process models of all subprocesses and integrate them in a global process model. If this subprocesses extraction has finished it is necessary to perform all other CTPs.

This approach is limited since depending on the complexity of the overall resulting process model, with special respect to the nesting deep of its control-flow blocks, this might result in potentially too complex overall user interfaces. Thus to get the possibility to generate new complex user interface instances for the subprocess blocks a UI Model FormTabTemplate has been generated by the CTP5 as marker element for each subprocess control-flow block. The reason for this is to keep the information that these elements belong to a certain subprocess in the UI Model and if the nesting deep of control-flow blocks reaches a certain limit create new UI Model instance for each subprocess. With the help of the marker FormTabTemplate it is now possible to link the UI Model instances of the subprocesses with the global one.

During the execution of a process, which contains subprocesses the generated global complex user interface would be hidden, if a subprocess placeholder FormTabTemplate element is reach in the control-flow. Instead, a new complex user interface based on the process model of the subprocess would be displayed. If the subprocess instance has finished the global user interface would be displayed again with the state it was before leaving it for the subprocess execution. And according to the state of the UI Model after the FormTabTemplate placeholder element for the subprocess.

5.4.2 Behaviour Block Transformation

As mentioned already previous research has shown up that there exist process models which require special treatment according to the generation of respective user interface elements [Kam11]. Analyzing of several process models according to their required user interfaces during this work supplement this demand (cf. Chapter 4.1). In this sense, a process model is an arbitrary set of activities. In the samples presented in the following these models are a well-defined combination of human and services actives including their associated data-flow. Such a process model is called Behaviour Block in the following, since it links a set of process model elements to a set of user interface elements including a desired behaviour.

Table 5.15 shows the details for an example of a Behaviour Block Transformation Pattern, called *CTP6: Background Activity Transformation.* It defines the circumstance that it is necessary to load data elements by a service activity between two human activities. This data loading activity is triggered by the first human activity, performed by the service activity and the results of this data loading are displayed by the second human activity. The term background refers to the fact that in a corresponding user interface the data is loaded if passing from the UI screen of the first activity to the UI screen of the second activity in the background.

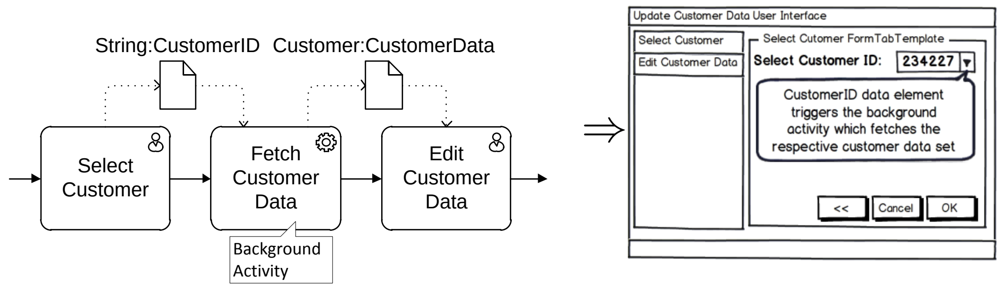
CTP6: Background Activity Transformation	
Description:	While human task activities are executed none human task activities can be executed in the background.
Example:	An agent selects a customer name in a form for editing customer data. After the agent has selected the name all other customer data fields in the form should be filled with the data of the customer loaded from a CRM system.
	 String:CustomerID Customer:CustomerData Select Customer Fetch Customer Data Edit Customer Data Background Activity ⇒ Update Customer Data User Interface Select Customer Edit Customer Data Select Cutomer FormTabTemplate Select Customer ID: 234227 CustomerID data element triggers the background activity which fetches the respective customer data set << Cancel OK
Problem:	Human task activities, which are connected to a background activity, should result in a dynamic form. Dynamic refers to the automated loading of data and the automated displaying of new form elements in the context of a user interface.
Prerequisite:	Detection of non-human task (service) activities, which are background activities.
Implementation:	Dynamic forms, which contain background activities, need a change listener mechanism to detect user inputs a react on these inputs.
Related Patterns:	ETP1, ETP2, CTP1

Table 5.15: CTP6: Background Activity Transformation

Another example for such a process model snippet with additional semantics according to user interface generation is shown in Figure 5.6. It covers the quite common aspect of editing existing data by a human resource. It consists of three activities whereupon the first and last activities are service activities. In the first service activity existing data is loaded e.g., from an external storage system. In the second activity, this data is displayed to a user with the option to edit the single data elements by providing an adequate user interface. If this edit activity is finished by submitting the user interface form the updated data values are stored by the execution of the subsequent service activity.

The last example of a process model that can be handled standardized according to its user interface related behaviour is shown in Figure 5.7. It is a semantic extension to the previously presented *CTP4: Loop Block Transformation* (cf. Table 5.13). It handles the

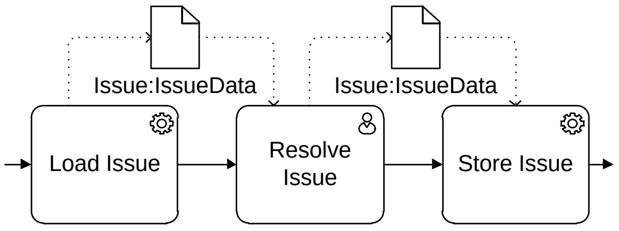


Figure 5.6: Edit Activity Process Model

aspect that the input of data in a form has to be repeated until the input data is corrected according to predefined validation criteria. The generated user interface elements can be build by the previously defined CTPs and ETPs. The addition of this pattern is the semantic according to quitting the loop. This is triggered by a service activity at the end of the loop. The service activity uses the data elements generated by previous activities inside the loop block to trigger the decision for quitting the loop.

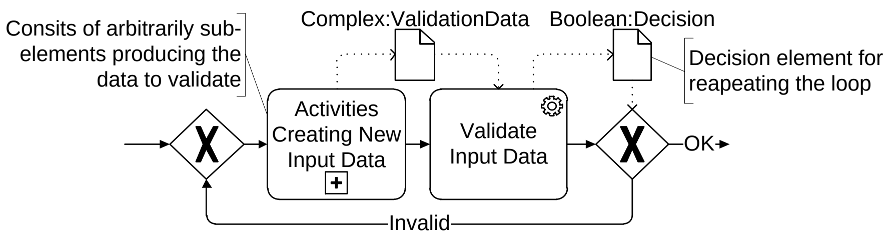

Consits of arbitrarily sub-elements producing the data to validate
Complex:ValidationData
Boolean:Decision
Decision element for reapeating the loop
Activities Creating New Input Data
Validate Input Data
OK
Invalid

Figure 5.7: Validate Activity Input Data Process Model

The actual presented three behaviour block transformation CTPs are just a starting point for more detailed research into the direction of semantic process model interpretation, according to complex user interface generation concerns. A starting point to identify and classify such constructs, would be a detailed analyze of existing process models from various domains according to their corresponding user interfaces.

In this chapter a Transformation Model has been introduced as a framework for an overall process model to user interface transformation. The Transformation Model takes care of separating the different concerns according to complex user interface generation in distinct components (parts). Moreover, it includes a description of how the single components are connected with each other. Two of the Transformation Model components have been introduced in this chapter. One the one hand Elementary Transformation Patterns, which describe the basic aspects of how to handle process model activities and their respective data flow according to user interface generation. On the other hand, Complex Transformation Patterns, which handle the process model control-flow structure transformation to User Interface Model elements. Additionally, the control-flow structure CTPs are supplemented with Behaviour Block CTPs, which describe User Interface Model element generation based on a interpretation of process model semantics.

Referring to the uses cases defined in 4.1 some of the complex user interface requirements are still not treated. The execution of the correct CTPs for each control-flow block of a process model including the nested execution of ETPs is not sufficient for the realization of an overall complex user interface generation. There are more advanced requirements especially according to the generation of role specific UIs and the interconnection of the single Transformation Model components. In addition, one should keep in mind that the Transformation Patterns only describe the generation of UI Model elements, which is not the same as a user interface. All theses aspects are addressed in the subsequent chapter, which handles role specific grouping, describes an overall UI Transformation Algorithm and last but not least specifies how a UI widgets for a complex user interface can be generated based on a UI Model.

6 Transformation Model Composition

The Transformation Patterns introduced in Chapter 5 are not sufficient to realize a complete transformation from a process model to all user interface elements needed to execute process instances. To realize this, rules for the application of the patterns are necessary. Additionally, the allocation of activities to be considered for the generation of a user interface instance, based on grouping criteria, has to be discussed. With the help of this role-specific user interfaces can be generated.

Therefore Chapter 6.1 discusses the way of grouping activities and the influence of this grouping according to the overall process model transformation. Chapter 6.2 introduces a Mapping Model as basis for the generic process model to user interface Transformation Algorithm. The algorithm applies the Patterns of Chapter 5, the Activity Grouping Mechanism and considers the logical process steps to describe the generation of end-user interfaces from process models. Furthermore, the actions that are necessary to delegate changes from a user interface to a process model and vice versa are specified.

6.1 Activity Allocation

In this chapter all important aspects of allocating activities during user interface generation are discussed. The term of allocating activities on the one hand refers to the action of considering activities for the generation of a certain user interface instance and others not. On the other hand, it describes the merge or split of activities to fulfill additional user interface requirements. Thus in the Chapter 6.1.1 the most important aspect, a role model used by a process model, for grouping activities is introduced. This is followed by Chapter 6.1.2 which discusses implications about the detail levels of process models supplemented by an approach to countervail this detail level differences and concludes with necessary requirements for process models to achieve useable results by a user interfaces generation.

6.1.1 Role-based Activity Grouping

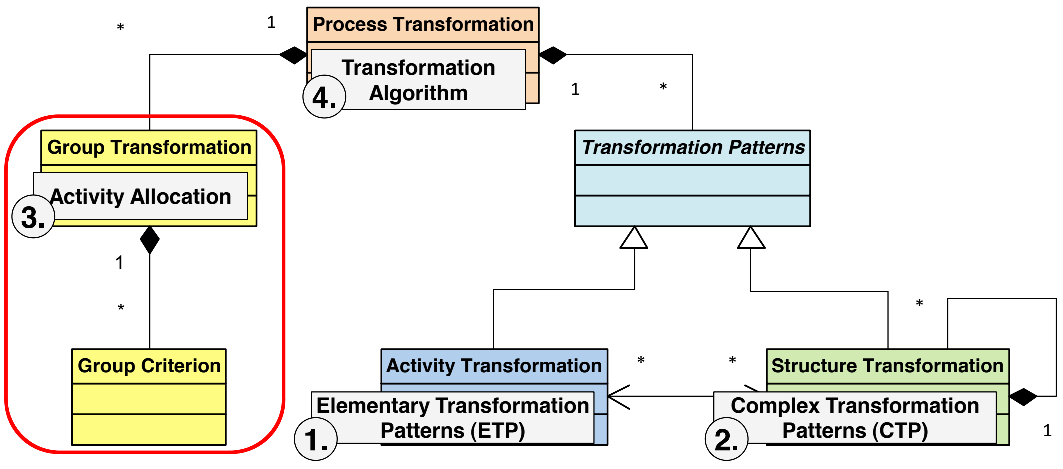


Figure 6.1: Activity Allocation in the Transformation Model

The grouping of activities is an important aspect during the overall end-user interface generation process. It has direct influence on whether an activity is taken into account for the creation of a certain user interface or not [ZZHM07]. Figure 6.1 shows the direct connection between *Group Transformation* and the overall *Process Transformation*. In the following, all aspects of Group Transformation are discussed in detail.

First of all a clear definition is required for the term *Group Transformation*. In this sense, Group Transformation covers the aspect that there are certain, later discussed, criteria influencing either the assignment or not assignment of activities to certain end-user interfaces. For this purpose Figure 6.2 shows the hierarchical structure of all aspects which are relevant to realize a Group Transformation as UML class diagram. The only thing not covered in Figure 6.2 is the aspect that a complete process model transformation can consist of multiple Group Transformations. This is covered by the aggregation between Group Transformation and Process Transformation in Figure 6.1. A Group Transformation, in turn, is based on an abstract Group Criterion. For this abstract Group Criterion, there are two essential characteristics to distinguish:

1. a **manual** defined **Group Criterion**, which can be seen as a fall back solution if the automated detection and processing of Group Transformations fails and
2. a **Group Criterion** either **elementary** or **complex** which is extracted from the activities of the process model.

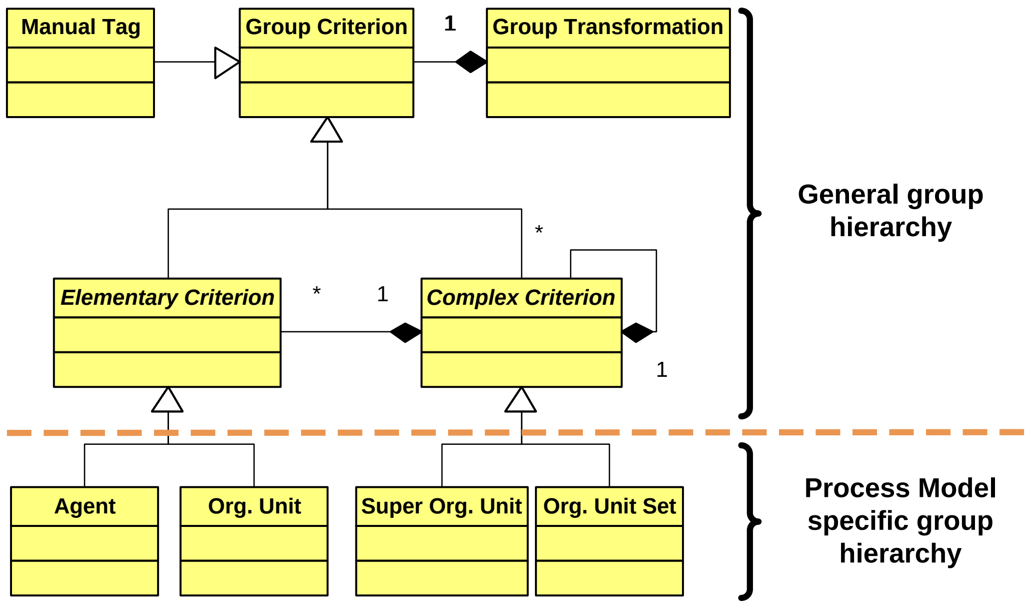


Figure 6.2: Group Transformation Hierarchy Structure

The special characteristics of Group Criteria depend on the specifics of the process model in use. In the following general characteristics, as shown in Figure 6.2, are explained. The base for the grouping model is the organizational model, which is used by the process model to transform. The application of abstract elementary or complex Grouping Criteria enables a simple but flexible grouping of process activities. Finally yet importantly this decouples the overall Group Transformation from a concrete process model implementation.

Based on this, there are two elementary Grouping Criteria. The first of them is the **Agent**. The term agent refers to a specific person, which is needed to execute a certain activity, e.g., only Mrs. Doe the CFO is allowed to confirm orders which have an amount above one Million Dollar. The second of them is the **Org. Unit** (Organizational Unit). This Group Criterion refers to the issue that the human resource executing an activity has to be part of a certain organizational unit, e.g., an agent has to be a member of the shipping department to confirm the shipping message send to a customer.

As shown in Figure 6.2 a complex Group Criterion can consist of multiple elementary Group Criteria. Furthermore, it can be composed by multiple different complex Group Criteria. In addition to its grouping nature based on elementary Grouping Criteria, there are two special characteristics of the complex Group Criterion explained in the following. The **Super Org.**

Unit (Super Organizational Unit) is a parent organizational unit for basic organizational units as mentioned in the elementary Group Criteria case. An example of a **Super Org. Unit** could be a production department, which consists of several child departments, like an order department, a development department, a shipping department, etc. The **Org. Unit Set** Group Criterion is used to express that multiple organizational units are used to realize a Group Transformation, but instead of being in a hierarchical order like in the **Super Org. Unit** case, the underlying departments can be completely indecent form each other.

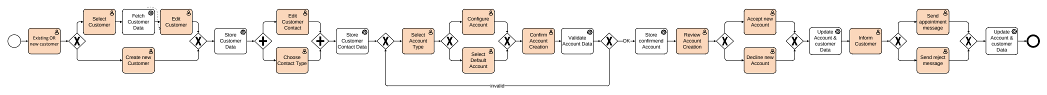
After this introduction of the different Group Criteria characteristics now, the application of these criteria in the overall end-user interface generation process will be exposed. This application is called **Group Transformation** in the following (cf. Figure 6.2). It tries to allocate the activities of a business process model, according to their Group Criterion characteristics, to certain end-user interfaces templates. In which a user interface template is the resulting generated UI-based on a User Interface Model and a user interface instance is the instantiation of such a user interface template during process run-time.

These templates act as an input basis for the Transformation Patterns introduced in Chapter 5. The main purpose of the Group Transformation in our approach is to realize the generation of user role specific user interface instances. Details for this UI instances generation are discussed in Chapter 6.2.4. As outlined e.g. in [ZZHM07] role specific user interfaces are important based on the aspect of information overload and privacy. This results in presenting only this kind of information to a user, which are relevant to perform the role specific actions.

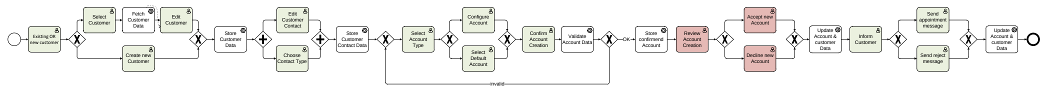
Thus, in the following the realization of a Group Transformation for user role-specific UI instances is described. This transformation is separated in four overall processing steps. Figure 6.3 illustrates theses steps, which are described in the following:

1. **Group Criterion Detection:** Detect all Human Tasks in a process model and their respective Group Criterion. In Figure 6.31 the Group Criterion is predefined as 'Organizational Unit'. This results in separate User Interface templates for each Org. Unit found in the process model. Instead of using a predefined Group Criterion, it would be possible to extract all valid Group Criteria from the Process Model and suggest them to a user for selection.

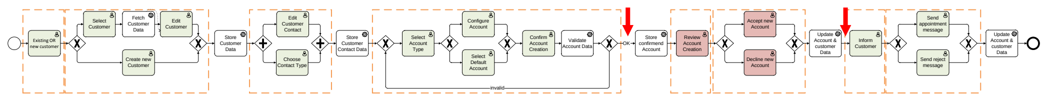
2. **Group Criterion Composition:** Create a mapping from distinct Group Criteria found in the process model to the corresponding activities. In Figure 6.32 one can see that the human activities found in the previous step have been grouped by color



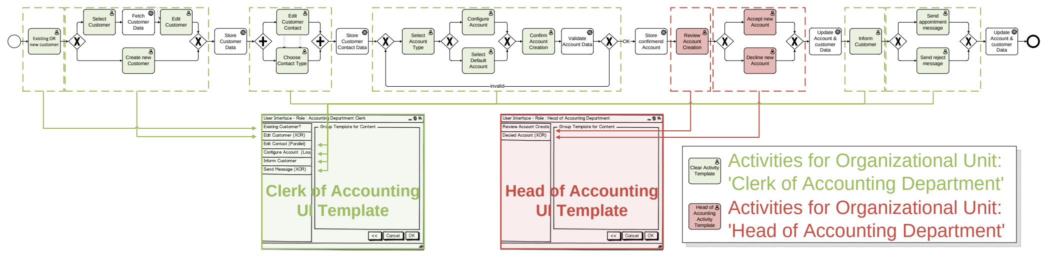
(1) Selection of all Human Activities (orange highlighted), predefined Group Criterion is 'Organizational Unit'



(2) Human Activities grouped by their executing Organizational Units (roles)



(3) First Level of detected control-flow blocks relevant for User Interface generation


Clerk of Accounting UI Template
Head of Accounting UI Template
Activities for Organizational Unit: 'Clerk of Accounting Department'
Activities for Organizational Unit: 'Head of Accounting Department'

(4) Allocation of control-flow blocks and the respective activities to role-specific User Interface templates

Figure 6.3: Steps of Group Transformation

highlighting. The green colored activities are linked to the Org. Unit 'Clerk of Accounting Department' and the red colored activities are linked to the Org. Unit 'Head of Accounting Department' instead.

3. **Process Model Aggregation:** Based on the Group Criterion Composition for each Group Criterion a separate Process Model is created. In each of the Group Criterion specific process models, the intersection points with other Group Criterion specific process models have to be stored. This is necessary to enable a multi-role-specific user interface for a collaborative process execution. If a certain intersection point is reached, an information message has to be send to all roles participating in the execution of the process instance. Figure 6.33 shows the intersection points for the sample case process model including all detected control-flow blocks which are relevant for successive transformation steps.

4. **Transformation Execution:** All control-flow blocks and the respective activities belonging to a certain Group Criterion are mapped to one end-user interface template. This is realized using the Group Criterion-specific process models from the previous step. For each of this Group Criterion-specific process models the transformation to a user interface template is executed. This action can be summarized by, detecting all control-flow blocks followed by the application of the Transformation Patterns (CTPs and ETPs) previously introduced. The exact details about the execution of the overall transformation are described in subsequent Chapter 6.2. Figure 6.34 shows the Org. Unit-Specific user interface templates, which are the results of executing the overall process model to user interface transformation.

Another important aspect, according to the allocation of process model activities to end-user interface templates, is the separation of the model in logical execution steps. This might result either in an additional merge of activities or even in a split of single activities according to their end-user interface template allocation. This secondary fragmentation of the overall process model has to be taken into account before defining the overall process model to user interface transformation. Therefore, the following chapter outlines the so-called *Granularity Problem* followed by an approach how to deal with this logical process execution steps in respect of the overall end-user interface generation process.

6.1.2 Variability of Process Granularity

The detail level of a process model depends on different parameters [LK01, RM08, HRL09]. One of them is the application domain of the process model [EKO07]. This results in a wide range of different model types, which reaches from process models which are build for documentation only with the help of word processing office software, to detailed technical models specified in process languages like BPEL [OAS07], ready for execution in BPMS. Even if the process model definition environment and the usage of the model is well-defined, like in our case, the developed models distinguish in a wide range according to the granularity of activities and the respective modeled data objects [EKO07, MSBS03]. Such different detail granularity often depends on the background of the user modelling the process model [GVGW08] and on the process modelling environment setup in use [MS08].

The different detail levels of process models result in implications for the user interface generation. To handle this different detail levels during end-user interface generation process an abstraction mechanism is needed to map process model activities to certain user interface

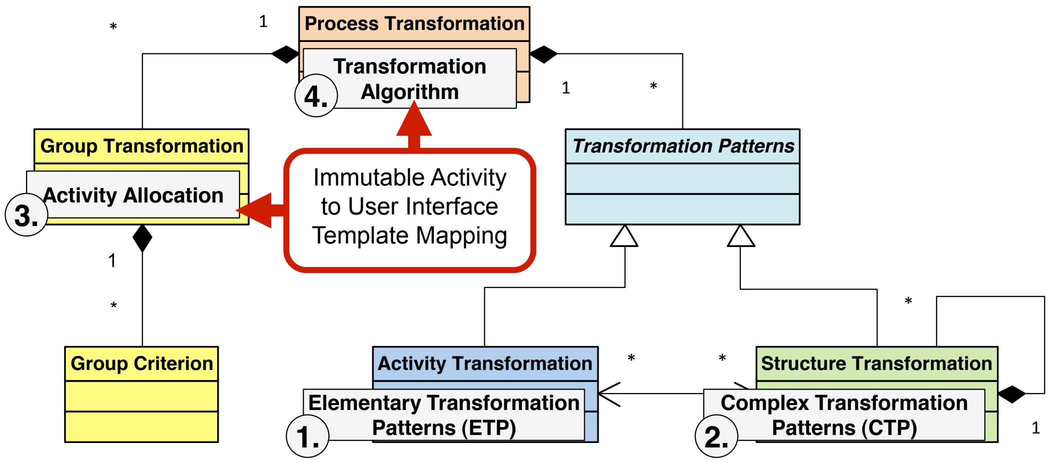
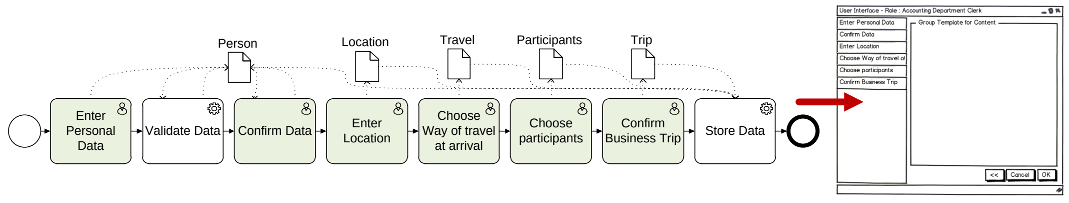


Figure 6.4: Location points in the Transformation Model for Template Activity to User Interface Template Mapping

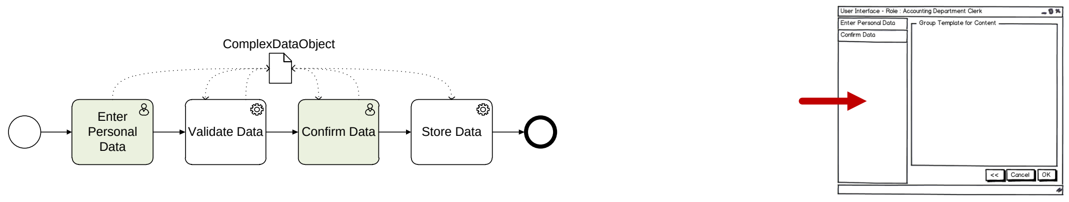
templates in an immutable manner. This mapping should on the one hand be independent from the transformation process steps described before, but on the other hand extend them with the required capabilities as needed for desired results during user interface generation. The immutable activity to user interface template mapping has to be defined before the application of the Transformation Patterns. The reason for this is to avoid undesired allocation of activities based on the default interpretation of process model control-flow because of the Transformation Pattern application. Figure 6.4 illustrates possible location points for the creation of an immutable activity to user interface template mapping in the overall Transformation Model.

For a better understanding of the outlined Granularity Problem Figure 6.5 shows two process models with different granularity levels. Figure 6.5b has insufficient distinct activities for a resulting user interface with good usability whereas Figure 6.5a has too much distinct activities. In this sense, there are two different cases according to the activity detail level of process models, which should be considered while developing a Transformation Model. They are described in the following:

1. **Too much details according to modeled activities:** The example process model in Figure 6.5a shows an example of a process model with its six for user interface generation relevant human activities highlighted. A simple application of the Transformation Patterns would result in a user interface as shown on the right side. For each single activity a FormTabTemplate element would be generated in the user inter-



(a) Process Model with too many activities, including resulting user interface



(b) Process Model with insufficient activities, including resulting user interface

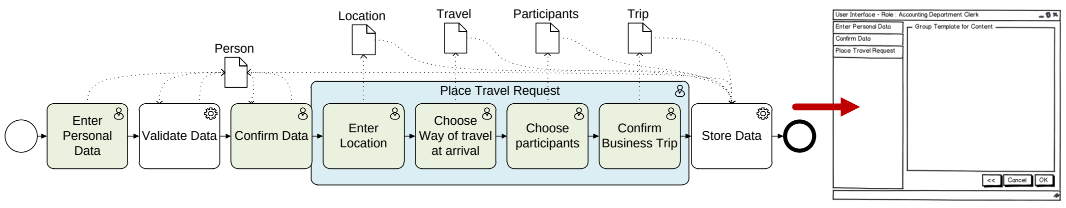
Figure 6.5: Granularity Problems in Process Models

face template resulting in six tabs overall. However, this is not the desired result. In the process model, the activities represent only primitive tasks. Thus they are to fine grained and result in a to complex FormTabTemplate set.

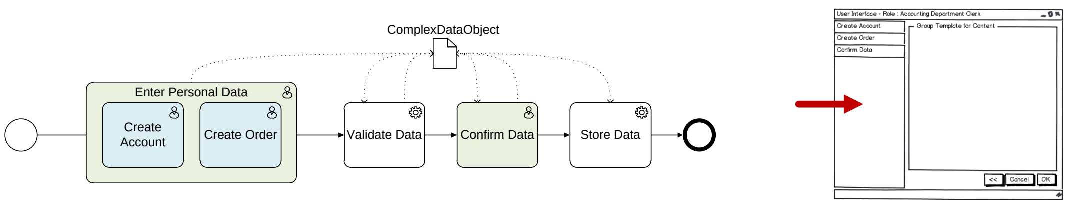
2. **Insufficient details according to modeled activities:** The process model example in Figure 6.5b shows an activity chain of two human activities. Thus, its transformation will result in a user interface template with two FormTabTemplate elements. However, this is not the desired behaviour, according to the complexity of the modeled activities. If the complexity of activities is high it might even be necessary to split a single activity in multiple ones to get the desired results for user interface generation.

The both outlined examples showed that for some process models additional, logical process execution steps, have to be taken into account for the user interface template generation. This might result either in additional grouping (*case 1. too detailed*) or in a splitting (*case 2. insufficient details*) of activities in respect to their allocation to, or even the creation of, certain user interface templates.

For this logical process execution steps in the following the term **task** is used. Such a task can be described as the amount of work done in a single step by one human resource during the execution of a business process instance, independent of the activities contained in the process model. In addition, the usage of the term task is conform according to its introduction in the Chapter 2.1, e.g., in [SMV10, PMM97] a task has been defined as the amount of work to reach a certain goal.



(a) Multiple activities grouped in a single Activity now resulting in a User Interface with three Tab elements



(b) Complex activity split into two Activities now result in User Interface with three Tab elements

Figure 6.6: Granularity Problem in Process Models solved by introduction of new Activities

Figure 6.6 shows a potential solution for the Granularity Problem example of Figure 6.5. This illustrates that it is necessary to arrange activities from the process model to parent activities (Figure 6.6a) or split activities into sub-activities (Figure 6.6b). These parent or sub-activities are the previously mentioned tasks. Tasks or so-called task modells [MPS02] represent a general approach to describe the logical activities to reach a certain goal in the field of Human Computer Interaction (HCI) [Pat00]. Moreover, as outlined in Chapter 2.1, task modelling is established to describe user interactions for model-driven user interface development.

For good user interface generation result for process models with different granularity level a method to decide about the grouping or splitting of activities is necessary. An obvious approach is the analyze of the amount of data objects (based on overall primitive data types) used by an activity, e.g. as proposed in [ZZHM07]. This would require the definition of a minimum and maximum amount (range) of primitive data objects processed in one screen of a user interface presented to a user. Based on the amount of metadata included in a process model this approach is realizable even in an overall complete automated process model to user interface transformation. However, it might not be sufficient to solve all the undesired results during a user interface generation. Therefore, additional actions are required.

Based on the *Manual Tagging* option of the Group Transformation approach, it is possible to provide a user with the capabilities needed to assign activities individually to user interface elements. Additionally required for this is the possibility to distinct between different kinds

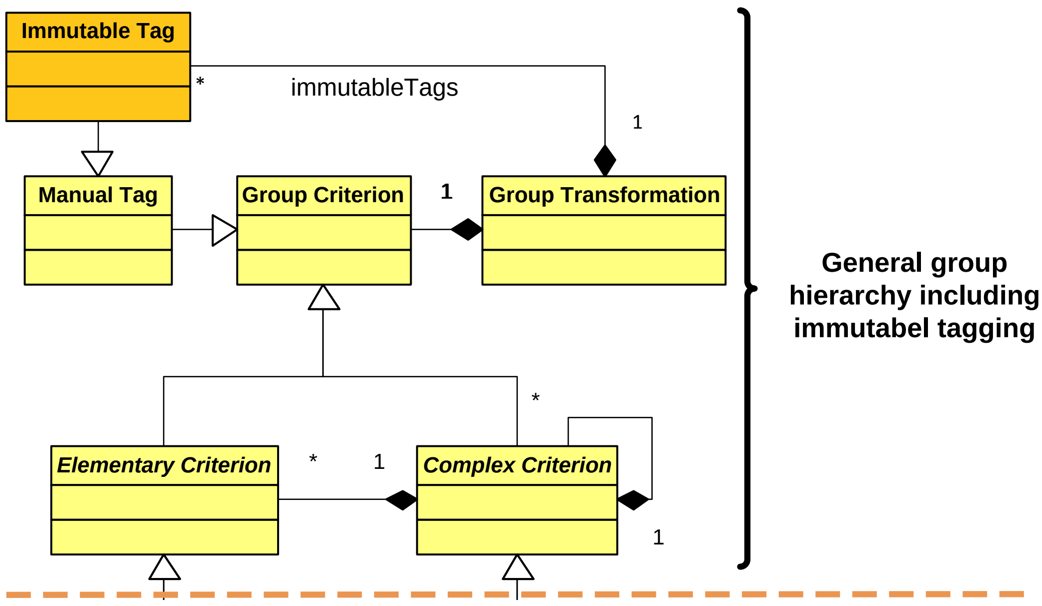


Figure 6.7: Manual Tag Group Transformation extended by Immutable Grouping Mechanism

of manual tags. The standard manual tag, which is a replacement for the overall Grouping Transformation and an additional manual tag, which can be used to mark activities as a immutable group. This additional immutable group manual tag has to be considered in the overall Transformation Algorithm as well. Figure 6.7 shows the relevant section of the Group Transformation UML class diagram extended with necessary immutable group manual tag mechanism. With the help of this it is possible to handle the previously outlined Granularity Problem and the Sequence Problem introduced in the Complex Transformation Pattern chapter (cf. Chapter 5.4.1).

Therefore Figure, 6.8 illustrates a possible solution for the Sequence Problem. Whereas the Sequence Problem is the default resulting user interface for a arbitrary sequence of human activities in a process model, e.g. as shown on the bottom left side of Figure 6.8. This is caused by the recognition as sequence control-flow block with six activities and the consequential transformation of each activity to a single FormTabTemplate and thus a tab element. This will result in more distinct tab elements as required for a user interface with well usability and force the user to numerous unnecessary interactions. Therefore, the sequence of six activities is tagged with two immutable grouping tags, which changes the generated user interface consisting of two tab elements instead of six.

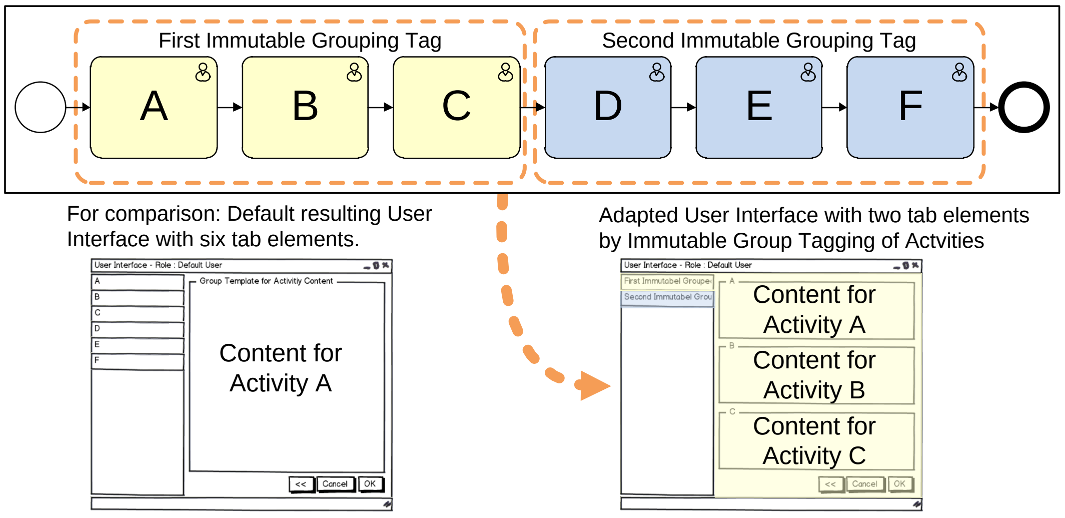

First Immutable Grouping Tag
Second Immutable Grouping Tag
A
B
C
D
E
F
For comparison: Default resulting User Interface with six tab elements.
Adapted User Interface with two tab elements by Immutable Group Tagging of Actvities
User Interface - Role : Default User
A
B
C
D
E
F
Group Template for Activitiy Content
Content for Activity A
<<
Cancel
OK
User Interface - Role : Default User
First Immutabel Groupe
Second Immutabel Grou
A
Content for Activity A
B
Content for Activity B
C
Content for Activity C
<<
Cancel
OK

Figure 6.8: Manual Tagging to Solve the Sequence Problem

The outlined problems and requirements to realize a sensible activity allocation independent and as a prerequisite for the process model control-flow structure transformation lead to some general limitations for the performance of the overall process model to user interface transformation. These limitations are summarized in the following.

6.1.3 Resulting Limitations

In the following prerequisites according to process model design to achieve good results according for usability of user interface generation are specified. Furthermore, the results of the activity allocation chapter are summarized.

First general point to mention about process modeling is the granularity of activities. For optimal user interface generation results the granularity level should be consistent in the overall process model. In addition, it makes sense to think about the required resulting user interface while creating a process model. Based on this, the human activities should be arranged in conjunction with the service activities and the control-flow blocks. For instance, it would be possible to arrange multiple sequential activities in a parallel control-flow block to merge them for their processing in a single screen (FormTabTemplate element) of the user interface.

Second, during the creation of a process model is the aspect of levels of nesting control-flow blocks. This should be limited to a reasonable amount of levels. Even though in theory it is possible to create and handle an endless nesting deep of control-flow blocks during user interface generation. The resulting user interface would be too complex and confusing for a user. However, it would be possible to generate new user interface instances from a specified control-flow block nesting level on and link them with the global user interface instance. This would in turn result in a much more complex handling during the overall user interface transformation.

Third, the creation of process models for good user interface results is the assignment of roles from an organizational model to single human activities. For this it is necessary to stay on the same hierarchy level based on the Group Transformation hierarchy introduced in Chapter 6.1.1. For example, if a single human activity is assigned by an agent all of the human activities should be assigned by agents as well instead of assigning them by an Org. Unit Set.

Thus, the conclusion about process modeling for user interface results with good usability is to keep anything in a consistent way in the complete process model. This includes consistent allocation of used role granularity, consistent granularity of activities and even a consistent nesting deep of control-flow blocks.

The grouping of process model activities based on metadata like an assignments resulting from an organizational model is an essential prerequisite for control-flow structure transformation. Indeed, it would be not possible to generate role specific user interfaces from a process model without a Grouping Mechanism. A crucial part in the course of grouping is the interconnection of single grouped user interface instances based on intersection points, especially during the execution of process instances based on generated user interfaces. For the overall generation of user interfaces, which are capable for the execution of process instances, requiring multiple roles the implementation of such an intersection point feature is mandatory.

As mentioned before a granularity level of activities equal for the overall process model will fit best for user interface generation. Different activity granularity can have effects on the performed grouping of activities. Nevertheless, to have the possibility to achieve acceptable results independent of activity granularity during user interface generation additional efforts are necessary. Two approaches for handling granularity differences have been presented.

The first possible solution could be the analyze of complexity of single activities, based on counting the processed primitive data elements. With the advantage, that such an approach can be integrated in a fully automated user interface generation process. However, the disadvantage that process and activity semantics are completely ignored. For instance, it might be possible that a process model and a possible resulting user interface rely on the different granularity levels of activities.

An alternative to the analysis of complexity approach is the presented manual immutable group tagging approach. This extends the manual tagging from the default Grouping Mechanism with the required capabilities to create immutable groups of activities. This in turn opens up the possibility to aggregate activities to complex user interfaces, which are capable for the processing of multiple process model activities in a single user interface screen, including the important aspect of making this immutable activity groups independent of the standard transformation. The major advantage of this approach is its very simple but flexible manner. With it a variety of (to be defined) problems including the outlined Sequence Problem, can be handled. The disadvantage is the requirement of manual user interference during a fully automated process model to user interface transformation.

With the help of this immutable group tagging approach, the previously introduced Grouping Mechanism and the Transformation Patterns (cf. Chapter 5), now it is possible to illustrate the overall Transformation Algorithm implemented in the Transformation Model. This Transformation Algorithm interconnects the single steps necessary for end-user interface generation to an overall transformation process. The following chapter describes this overall transformation process in detail.

6.2 Process Model to User Interface Transformation

In this chapter, the developed overall Transformation Algorithm from process models to user interfaces is presented. It applies the Transformation Patterns by a well defined manner of control-flow block processing and considers the activity allocation aspects according to grouping outlined in Chapter 6.1.1.

First, the definition of a so called *Mapping Meta Model* in Chapter 6.2.1 is presented. This Mapping Meta Model is a clear definition of how User Interface Model elements can be mapped to process model elements and vice versa. It serves as the base to perform an overall process model to user interface transformation. This is supplemented by Chapter

6.2.2, which presents basics according to the processing of control-flow blocks in the transformation. The algorithm to execute such a transformation is introduced and illustrated in Chapter 6.2.3. To support the bidirectional propagation of changes from a user interface template to a process model additional efforts and definitions are necessary, e.g., restricting the allowed change operations to achieve sensible user interface and process model results. Hence Chapter 6.2.4 deals with the aspects of how to delegate changes from one (model) side to another. Finally, Chapter 6.2.5 concludes with the application of complex user interfaces generated on the basis of the use cases (cf. Chapter 4.1).

6.2.1 Mapping Meta Model

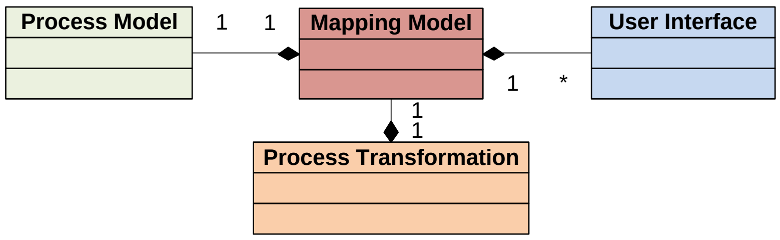


Figure 6.9: Interconnection between Process Model, User Interface Model and Process Transformation by the Mapping Meta Model

The Mapping Meta Model is based on the block-oriented process model introduced in Chapter 3.1 and the User Interface Model introduced in Chapter 3.2. As illustrated in Figure 6.9 it serves as an interconnection (meta model) between those two models. It is used to map process model elements with respective User Interface Model elements during the handling of the process model while performing the Transformation Algorithm. To realize the needed mappings it is separated into four hierarchy levels. Table 6.1 gives an overview of these mapping levels including the effected elements from both sides and a potential representation for the User Interface Model elements. The four hierarchically subdivided mapping levels are named after the respective entities they link with each other. In the following each of these mapping levels are described in detail.

Level 1: Process Model ⇔ User Interface On the first level for each processed process model respective User Interface Model elements are created. The fact that based on grouping multiple User Interface Models are required for a single process model has to be considered

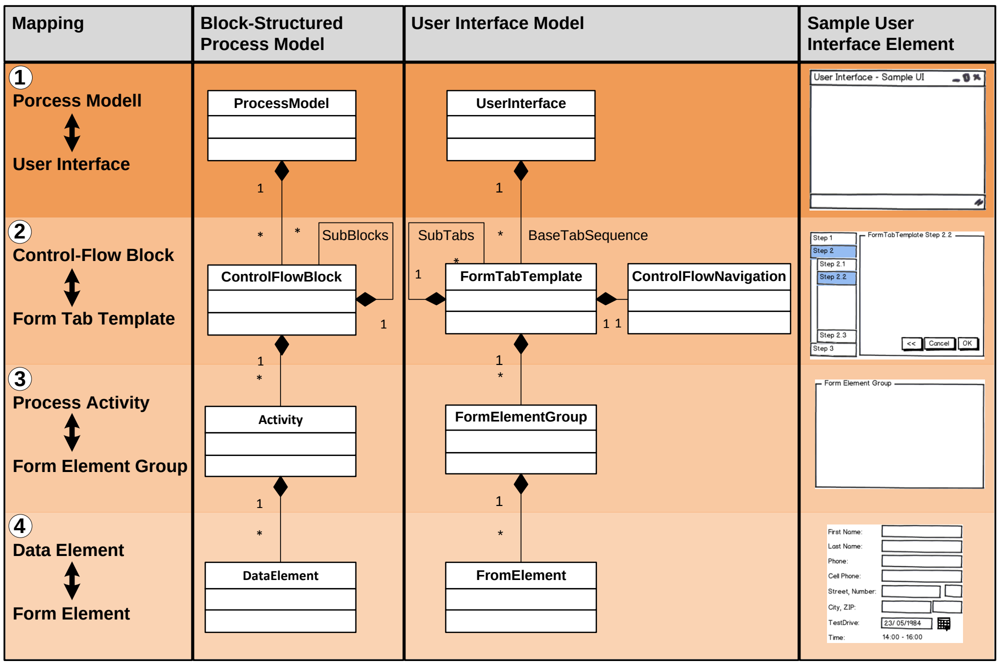


Table 6.1: Links between Hierarchy Levels of Block-oriented Process Model Elements and User Interface Model Elements

in a mapping model implementation. The default use case for this is that one process model is linked to a set of role specific User Interface Models. Base on these role-specific UI Models a role specific generation of a concrete user interface instance is possible.

Level 2: Control-Flow Block ⇔ Form Tab Template The second hierarchy level and all its subjacent hierarchy levels are used to fill the base User Interface Model element created on the first level with content elements that describe forms and form elements used for the generation of user interfaces, for handling process instance based on the respective process models. Consequently, on the second level container elements for the single steps while executing a process, the so-called *FormTabTemplates* (cf. Chapter 3.2.2) are generated. For each control-flow block of the process model, a FormTabTemplate is generated. This FormTabTemplate element can consist of the contents based on the transformation of single or multiple activities, or of nested FormTabTemplate elements (called *SubTabs* in Table 6.1) that are generated for nested control-flow blocks.

In addition, each FormTabTemplate element is connected with a *ControlFlowNavigation* element. This ControlFlowNavigation element is used to link FormTabTemplates based on the control-flow of the process model with each other. Based on this control-flow information, or in other words what is the next and previous step according to the process model, concrete user interface elements can be linked with each other. This, in turn, enables the generation of user interface instances with the possibility of bidirectional navigation between each displayed screen. The details for this navigation options are rather complex and therefore discussed in Chapter 7.3.

Level 3: Process Activity ⇔ Form Element Group On the third hierarchy level, for each activity of a process model control-flow block, a *FormElementGroup* User Interface Model element is generated. The FormElementGroup is allocated to a FormTabTemplate based on the allocation of the underlying activity to a control-flow block in the process model. By the nature of a control-flow block, multiple activities and therefore multiple FormElementGroups can be assigned to a FormTabTemplate. For example, if a parallel control-flow block consists of multiple human activities which have the same Grouping Criteria the generated FormElementGroups will all be associated with a single FormTabTemplate element. This in turn will result in a user interface instance containing multiple FormElementGroup elements.

Level 4: Data Element ⇔ Form Element The fourth level of the Mapping Meta Model is responsible for the generation of *FormElements* of the UI Model based on the data elements used by an activity of the process model. FormElement is an abstract base type for concrete input and output elements of a form. By analyzing the data-flow of an activity and the distinction between input and output data elements as it is realized by the Elementary Transformation Patterns (cf. Chapter 5.3), it is possible to generate a complete User Interface Model data structure from a process model.

Since the Mapping Meta Model is a model that describes how to transfer from one model (process model) to another (UI Model), in the terms of model-driven development, it is a meta model. The general operation purpose of the Mapping Meta Model during execution of the transformation is the generation of UI Model elements while respective process model elements are handled in the algorithms implementation. The result of executing a process model to User Interface Model transformation is a fully instantiated UI Model. This fully instantiated User Interface Model serves as an abstract description for the creation of a single or even multiple concrete user interface instances. It can be used to generate presentations of any kind, e.g., a HTML presentation for a web-based user interface. Subsequently it is possible to describe the overall Transformation Algorithm with all its details using the currently described Mapping Model as base data model. Therefore, the next Chapter starts with some basics according to the transformation of control-flow blocks.

6.2.2 Control-Flow Block Processing Basics

All process model control-flow block elements belong to one or more parent block elements, since it is possible to nest block elements with each other. This nesting aspect is address by a recursive application of the CTPs. The Transformation Model covers this by a composite like aggregation association between single CTP instances. The control-flow block elements are transformed starting with the outmost block element of the nested control-flow block elements. Thus, Figure 6.10 visualizes the basic aspects of the control-flow block structure transformation order based on a small process model example.

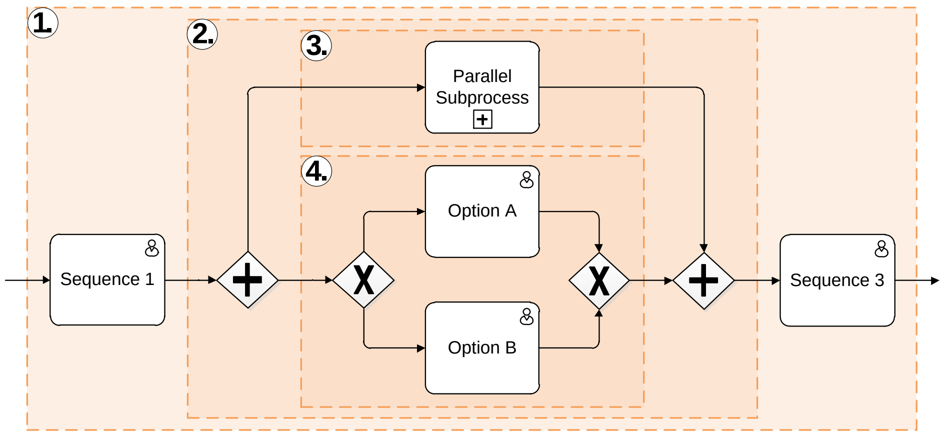


Figure 6.10: Processing Order of Nested Control-Flow Block Elements

In this example, there are four nested control-flow block elements. They are processed according to their numbering. The first detected control-flow block element is a sequence of three sub elements. In this sequence control-flow block, the start element (activity *Sequence 1*) and the end element (activity *Sequence 3*) are activities. The second element of the sequence control-flow block (**2.**) is a parallel control-flow block element. This parallel block in turn consists of two control-flow branches. The control-flow block building the upper branch is a subprocess control-flow block (**3.**). The lower branch (**4.**) contains a XOR control-flow block and is the last detected control-flow block. This XOR control-flow block in turn consists of two control-flow branches (execution paths). Each of the branches contains a single activity (respectively *Option A* and *Option B*).

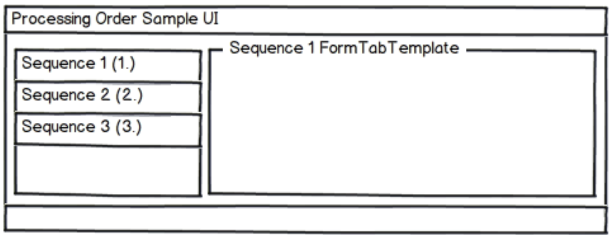
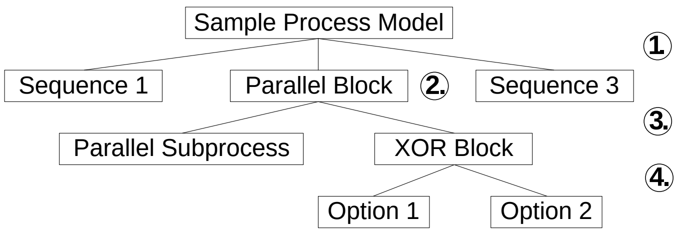
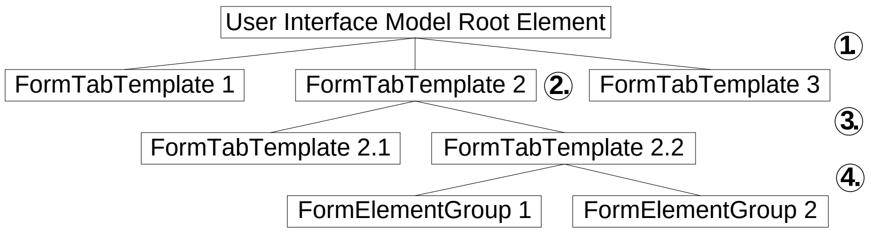


Figure 6.11: Resulting UI Screen after processing the Process Model in Figure 6.10

Figure 6.11 shows the resulting UI screen mock-up for processing of the sequence block elements. According to the UI Model, each of the sequence control-flow block elements results in a FormTabTemplate element arranged by its parent elements, e.g., the global UserInterface element. The association of the hierarchical tree-oriented data structures between the process model and the UI Model is visualized in Figure 6.12.



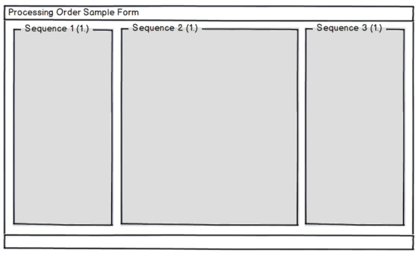
(a) Process Model control-flow block structure Hierarchy



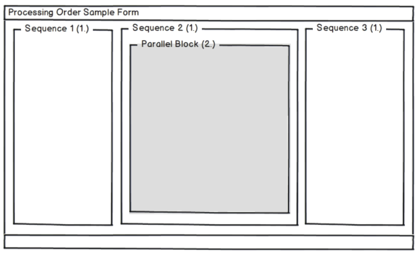
(b) UI Model Hierarchy

Figure 6.12: Comparison of Process Model Blocks and UI Model Element Hierarchy

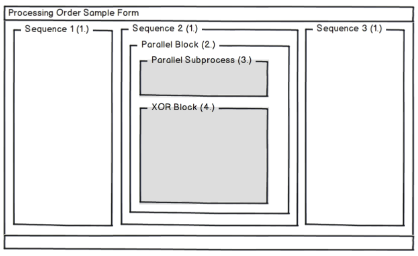
As introduced in Chapter 6.2.1, each of the process model elements can be mapped to an UI Model element counterpart. For a better understanding of how nested block elements must be processed according to the creation of UI Model elements and how a corresponding UI Widget creation could look like Figure 6.13 visualize the four steps of container element (FormTabTemplate & FormElementGroup) generation for the example process. For a better usability, all contents are displayed in a single user interface. The visualization of the four steps can be mapped to the levels of the hierarchical tree data structure of the UI Model. In which step 2 in Figure 6.13d of the visualization has no direct counterpart level in the tree since it is implemented by replacing the otherwise empty second FormTabTemplate of the Sequence control-flow block. With the help of these basics about the processing of



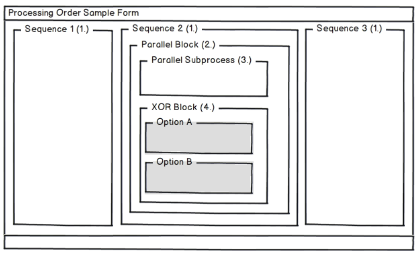
(a) Step 1: Sequence block



(b) Step 2: Parallel block



(c) Step 3: Parallel block contents



(d) Step 4: XOR block contents

Figure 6.13: UI Widget Mock-up Generation Steps for Process Model Control-Flow Blocks

process model control-flow blocks, it is now possible to define the overall process model to user interface Transformation Algorithm.

6.2.3 Transformation Algorithm

The algorithm presented in this chapter describes the overall transformation from a business process model to complex user interfaces. The resulting complex user interfaces are enabled for executing of respective process instances. Therefore, it connects the single parts for user interface generation as already presented. Namely, it combines the grouping aspects with the application of Complex and Elementary Transformation Patterns. The result of the transformation is a fully initialized User Interface Model. This User Interface Model is the base for the generation of a concrete user interface implementation. Thus, the presented algorithm is completely independent of any user interface implementation related technology. Figure 6.14 illustrates the single parts realizing the Transformation Algorithm and their nested call hierarchy. In the following a short overall introduction for the algorithm is given.

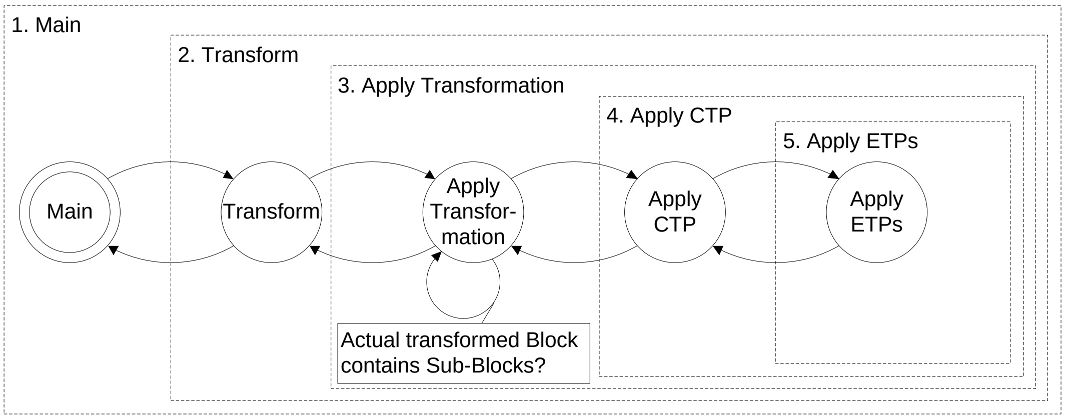


Figure 6.14: Overall Transformation Algorithm

Figure 6.14 shows that the algorithm is separated into five parts or modules. The **Main** module is used as a general setup mechanism, e.g., for setting up the process model as need for the latter transformation and a definition of the grouping to be used. Additionally, it is responsible for drawing a resulting User Interface Model. The Main module calls the **Transform** module. In the Transform module groups are extracted from the process model and based on this extracted groups for each of them a group-specific user interface is generated. This is done by creating an initial group-specific User Interface Model and calling the **ApplyTransformation** module. The ApplyTransformation module is recursively called as long as the actual processed control-flow block of the process model contains more nested control-flow blocks. In it for each processed control-flow block, the respective User Interface Model elements are generated. The generation of these UI Model elements is realized by fetching and **Applying** the fitting **Complex Transformation Pattern (CTP)** for the actually processed control-flow block. In addition, after the CTP application the processed block is checked whether it contains activities. If this is the case for all activities inside the block the **ETPs** are **Applied** for each of them. After this short abstract of the overall algorithm in the following, the single distinct modules are presented in detail.

1. Transformation Algorithm: Main Part As mentioned, the purpose of the Main module is the performance of the setup required for a process model to be enabled for performing the execution of the process model to user interface transformation and for drawing the results of the transformation. Listing 6.1 shows the single steps necessary for this in a pseudo code.

Listing 6.1: Process Model Transformation Algorithm *Main* Module

```
1 Main (ProcessModel pm)
2   ProcessModel pm = expandProcessModel(pm)
3   Grouping grouping = defineGrouping(pm)
4   Map<Group,UserInterface>res = Transform (grouping, ProcessModel)
5   for (UserInterface ui : res.values())
6     Representation rep = createRepresentation(ui)
7     rep.draw()
8 // Main END
```

The transformation setup starts with passing in a process model instance as input. This has to fit the process model definitions outlined in Chapter 3.1. In line two the process model is expanded. This includes the extraction of all contained subprocesses and their subsequent inclusion in a single hierarchically flattened process model. In line three of Listing 6.1 the Grouping Criteria as described in Chapter 6.1.1 is chosen. This can either be done automatically based on the Organizational Model entities linked with the human activities of the process model or manually by a user. Thus, this is also applied when the immutable groups definitions outlined in Chapter 6.1.2 occur. If activities should be allocated to an immutable group these activities of the process model are tagged respectively. After this two basic setup steps in line four the Transformation module is called passing over the expanded process model instance and the Grouping Criteria. The results of a Transformation are Grouping Criteria specific User Interface Models.

The subsequent lines of Listing 6.1 handle the processing of the Transformation results. If the call to the Transformation module returns resulting User Interface Models, for each of these Models a Representation is created. A Representation is an abstraction mechanism to take care of creating a real user interface including all necessary widgets based on a User Interface Model and to delegate UI-based changes to an underlying process model. More details about Representations are given in Chapter 6.2.4. The last call in the Main module takes care for the initial drawing of the Representations created based on the User Interface Models.

2. Transformation Algorithm: Transformation Part Listing 6.2 shows the second module that performs the general group specific setup for the performance of a process model to user interface transformation. It is started with a Grouping Criteria and a process model as input. These inputs are assigned by the call of the Transform module in the previous described Main module. In line two of Listing 6.2 all Groups based on the defined Grouping Criteria are extracted from the process model. For each of the extracted groups a User Interface Model is created. This is done by initializing a User Interface Model instance with group specific data, e.g., the name of the group is used as name for the User Interface Model. This is followed by extracting the group-specific process model and fetching the base control-flow block (cf. Chapter 3.1.1) from this group specific process model. With the initialized User Interface Model, the group specific process model and the base control-flow block the subsequent *ApplyTransformation* module is called. The result of the call to ApplyTransformation is stored in a map, which is returned to the Main module after all groups extracted from the process model have been processed.

Listing 6.2: Process Model Transformation Algorithm *Transform* Module

```
1  Transform (GroupingCriteria grouping, ProcessModel pm)
2    List groups = getGroups(grouping, ProcessModel)
3    Map<Group,UserInterface> result = new HashMap<Group,UserInterface>()
4    for ( Group group : groups)
5      UserInterface uiModel = createUIModel (group)
6      ProcessModel groupPM = fetchProcessModelForGroup (group, pm)
7      ControlFlowBlock base = groupPM.fetchBaseBlock()
8      ApplyTransformation (uiModel, groupPM, base)
9      result.put(group,uiModel)
10   return result;
11 // Transform END
```

3. Transformation Algorithm: ApplyTransformation Part Listing 6.3 shows the third module, which uses the hierarchically control-flow structure, stored in the process model to perform the application of CTPs and ETPs and thus in consequent the creation of a fully initialized User Interface Model instance. In line three of Listing 6.3 the respective Complex Transformation Pattern implementation for the actual processed control-flow block is fetched. For example, if the actual processed control-flow block is a parallel block the Complex Transformation Pattern implementation CTP2 is returned, which is aware of creating and or updating the necessary User Interface model elements. The application of the Complex Transformation Pattern is performed in line four. This application of the CTP instance in turn includes the application of all Elementary Transformation Patterns. The details for these Transformation Pattern applications are described in detail in subsequent chapters.

Listing 6.3: Process Model Transformation Algorithm *ApplyTransformation* Module

```
ApplyTransformation (UserInterface uiModel, ProcessModel pm,
  ControlFlowBlock block)
  CTP ctp = fetchCTP ( block )
  ctp.apply(uiModel)
  if( base.hasSubBlocks() )
    for (ControlFlowBlock subBlock: base.getSubBlocks() )
      ApplyTransformation (uiModel, groupPM, SubBlock) //recursion call
// ApplyTransformation END
```

Since the application of the CTP for each nested control-flow block (contained in the actual processed control-flow block) only creates empty place holder elements (defined as FormTabTemplate elements in Chapter 3.2.2) in the User Interface Model, after the application of the CTP the actual processed block has to be checked for sub-blocks. If the actual block contains any sub-blocks, for each of them the ApplyTransformation module has to be called reclusively with the sub-block as input parameter (cf. *recursion call* comment in Listing 6.3). This recursive call causes a full initialization of the User Interface Model, independent of control-flow block nesting deep in the process model. For a better understanding what happens in detail if the *Apply* method of a CTP implementation is called this is described in the following.

Listing 6.4: Process Model Transformation Algorithm CTPs *Apply* Method

```
Apply (UserInterface uiModel)
  // create UI Model elements for ControlFlowBlock, the abstract method
  // has to be implemented in each concrete CTP implementation for the
  // generation of UI Model elements as defined by the pattern
  ApplyCTP()
  if ( ControlFlowBlock.hasActivities() )
    for (Activity a : ControlFlowBlock.getActivities())
      ApplyETPs(a, uiModel) // create UI Model elements for Activities
// Apply END
```

4. Transformation Algorithm: ApplyCTP Part Listing 6.4 shows the most important part of the fourth module in the call hierarchy. This is implemented in the *Apply* method of CTPs abstract base implementation. It is responsible for the performance of UI Model modifications, which are defined in each CTP implementation. This includes the execution of all ETPs in correct order. The different CTP implementations have to take care for creating respective UI model elements, or update existing ones, based on the actual processed control-flow block. This is realized by the call to the *ApplyCTP* method in line five of Listing 6.4.

In the CTP implementation, only one nesting level is processed. This results in the creation of empty ForTabTemplate elements for a control-flow block, which is nested in another control-flow block in one iteration. In the following iteration, which processes the nested control-flow blocks, previously created ForTabTemplate elements are fetched from the UI Model and updated with the required content elements.

The *Apply* method in the CTP base implementation is responsible for the correct execution of the ETPs. Thus, in line six of Listing 6.4 the control-flow block processed by the CTP implementation is checked for contained activities. If the block contains activities for all these activities the *ApplyETPs* method is called with the currently processed activity and the UI Model instance as parameter. In the following chapter the important details about the *ApplyETPs* method are outlined.

Listing 6.5: Process Model Transformation Algorithm *ApplyETPs* Method

```
1 ApplyETPs (Actvitity a, UserInterface uiModel)
2   // human activity check
3   if(! new ETP1(a, uiModel).Apply() )
4     // non—human activity processing
5     new ETP2(a, uiModel).Apply()
6   else
7     // UI element generation
8     new ETP3(a, uiModel).Apply()
9 // ApplyETPs END
```

The Elementary Transformation Patterns (ETPs) are executed in the order as listed and numbered in the Chapter 5.3. This results in the ETPs execution chain as shown in Listing 6.5. In line three of this listing, ETP1 is used to check if an activity is relevant for UI Model element generation. The result of ETP1 defines if the activity is either a non-human activity or a human activity.

For non-human activities, source code that is required for connecting these activities to the human activities has to be generated. Since this source code generation is a rather complex topic for itself it is only handled in a limited way during this Thesis, e.g., in the Chapters 6.2.4, 7.2 and 7.3. The ETP implementation for *ETP2 Non Human Task Transformation* as defined in Chapter 5.3 encapsulates this source code generation (line five of Listing 6.5). For human activities an ETP3 instance is created and executed (line eight of Listing 6.5).

This includes the execution of all ETPs (ETP3.1 - ETP3.5) to generate UI Model elements for concrete input, output and edit elements.

After finishing the ETP execution by quitting the *ApplyETPs* method for all control-flow blocks of a process model, the overall initialization of the User Interface Model is finished. This User Interface Model can be used to generate a concrete user interface based on a certain specific UI technology. In Addition it is a important part for tracking changes made in a User Interface implementation to a process model and vice versa. These aspects are discussed in the subsequent chapters.

6.2.4 Propagating Changes

To enable the propagation of changes made in a generated user interface, to a process model and vice versa further aspects have to be discussed. In this chapter this tripartite aspects are described staring with an introduction in the following. The first part is build by the definition of new required components. The second part describes the mechanism how to propagate a change in both directions (from UI to process model), including an example. The chapter finishes with the suggestion of assistant user restrictions according to potential changes made in a user interface.

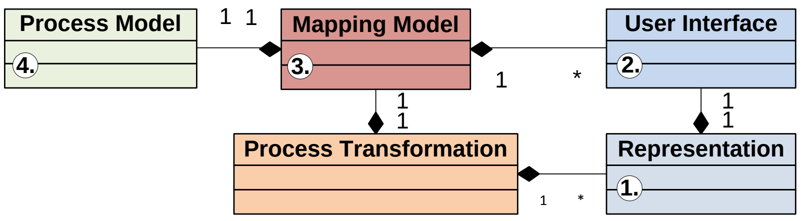


Figure 6.15: Change Propagation: From User Interface Representation to Process Model

Figure 6.15 shows the components required for the realization of change propagation. All components expect the Representation have been introduced in all details previously. As previously mentioned the Representation component is an abstraction mechanism for drawing User Interface Models. Thus, it is possible to implement different types of Representations, e.g., a web based implementation for a web browser user interface or a desktop implementation, e.g., based on Java Swing. This, in turn, leads to the fact that a Representation implementation manages all User Interface technology related things. In first place, it takes care of creating all required UI Widgets. In second place, it creates the possibility for delegating changes on UI Widget modifications to process model elements.

Since the creation of UI Widgets is very implementation technology related thing specific details about this are described in Chapter 8. Summarized, it can be outlined as a parsing and compilation of the previously created User Interface Model. In which parsing is the extraction of the single UI Model elements and compilation the iteration through these elements creating respective UI Widgets for each of the UI Model elements.

In the following, the details of the change propagation aspects are specified. The first requirement for delegating a change made in a user interface instance and thus, in turn, in a Representation is to know the affected UI Widgets. There are two cases to distinguish for this affected UI Widgets detection. The first case is the modification of an existing User Interface Representation element, e.g., changing a UI element that displays data to a UI element to edit data. In this case, the detection of the effected UI Widget is trivial since the UI Widget to change already exists and has to be selected by a user to trigger the change. The second case is the adding of new UI Widgets, e.g., adding a new input field to a form. In that case the first thing to do is the detection of the container UI Widget in which the new UI element should be added.

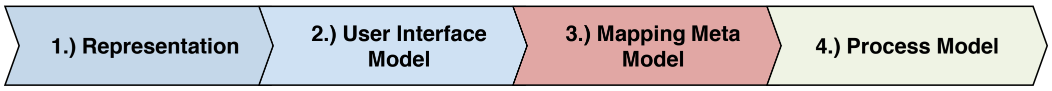


Figure 6.16: Steps of Change Propagation through Transformation Model Components

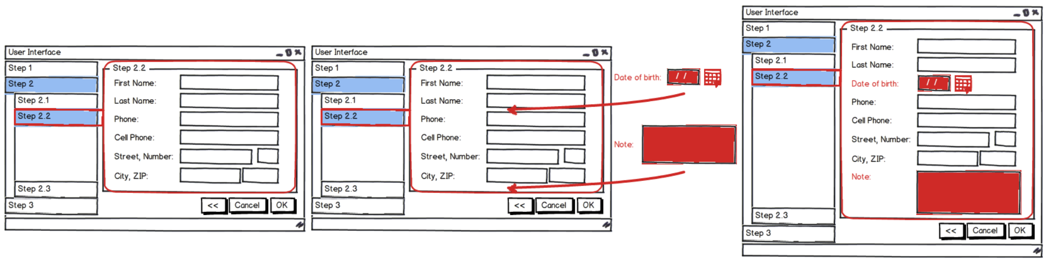
With help of the information, which UI Widgets are affected by a change it is then possible to use the Representation to get the underlying User Interface Model element in order to use the Mapping Meta Model to track the places in the process model, which are affected by user interface changes. Hence, it is a fundamental requirement for a Representation to keep the link between UI Widgets and UI Model elements. For a better understanding Figure 6.16 visualizes the path through steps of the single components required for delegating user interface changes to a process model. The following listing summarizes the purpose of the components in the case of change delegation in the order of their occurrence.

1. **Representation**, creates UI Widgets based on a User Interface Model and keeps the links for each single UI Widget to its respective User Interface Model element, detects the affected UI Widgets for changes in a user interface instance, creates new *User Interface Model* elements and updates existing ones

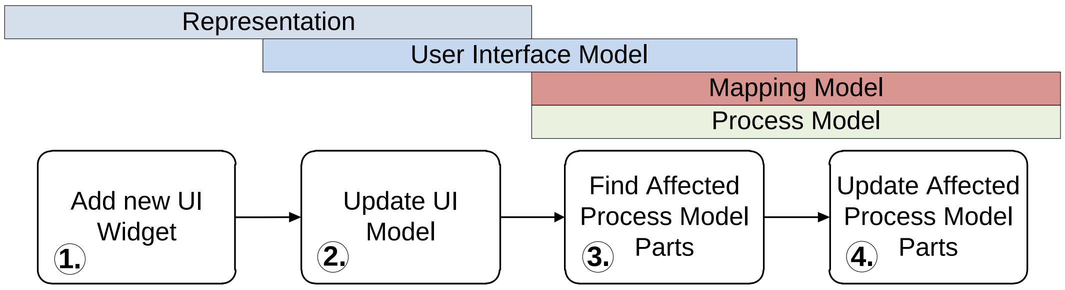
2. **User Interface Model**, holds all metadata required to generate a user interface instance and is used by the *Representation* to detect affected parts in a process model

3. **Mapping Meta Model**, links each *User Interface Model* elements with its corresponding *Process Model* elements and therefore enables the detection of affected parts in a process model
4. **Process Model**, describes the process order of activities and requires capabilities to check if a change triggered by a *Representation* (based on user interface change) is valid, it is used in the Mapping Meta Model to handle changes send from a *Representation*

This propagation procedure can be performed in reverse order from a process model to a Representation as well. To keep the implementation of such an reverse change propagation from a process model to a user interface as simple as possible it would make sense to re-execute the overall process model to user interface Transformation Algorithm as described in previous chapters and rebuild the underlying User Interface Model and Representation instances in the background.



(a) Change Propagation Steps in a User Interface (editor) to add new UI Widgets



(b) Change Propagation Steps and Component Involvement

Figure 6.17: Steps for adding new UI Widgets to an existing generated User Interface

Due to this implementation option for the propagation of changes from a process model to a user interface Representation the following explanations and definitions for change propagations are restricted to changes taking place in a user interface. Figure 6.17a shows an example for adding new UI Widget elements to an existing generated user interface instance. In this example, two new input fields should be added to a form for editing customer data. These two new fields are a field for the date of birth of the customer and a field for adding textual notes to a customer data record. In a user interface editor this new fields can be added by dragging and dropping this new elements to the existing user interface (template) instance. The user interface editor can highlight the affected parent UI elements and if the operation of adding new elements was successful display the new resulting user interface with the newly added fields highlighted. After a save operation for the updated user interface instance the highlighted elements would appear as normal like all other fields.

Figure 6.17b shows the corresponding operations including the involvement of the previously introduced components as color bars for the example presented in Figure 6.17a. In the first step, the new UI Widget element is created and added to the existing Widgets of the current screen. Based on the placement of the new UI Widget the corresponding parent UI Widget is detected. With the help of the new and the parent UI Widget elements it is possible to update the User Interface Model in a second step, since the parent UI Widget can be used to find the place in the UI Model for the newly added elements. This can be realized by a listener mechanism, which listens for changes in the user interface (template) or rather the Representation. This listener mechanism has to trigger the required User Interface Model changes proactively and delegate message like errors if a change made by a user is not valid according to process model restrictions to the Representation. After updating the UI Model it is possible to find the affected process model parts with the help of the Mapping Meta Model in a third step. In the fourth and last step, the respective process model parts are created and updated as needed. A validation if a change in a process model triggered from a user interface change is allowed has to be realized by the process model implementation. If the process model change validation results in an exception, the change in the user interface instance has to be disabled and an error message has to be displayed to a user.

The previous outlined example showed that a restriction of change operations made in a user interface (template) makes sense according to the avoidance of errors. Thus UI-based changes should be limited in a way to only allow such operations that are supported by the underlying process model. The question according to the limitation of change operations is which factors can be used for the calculation of a set of allowed operations? A starting point to answer this question is the classification of change operations not only to their kind (new or edit UI element) but in extend to the way they affect a process model. For this, we distinguish between local and extended UI-based change operations.

Whereas local changes are limited to a single screen of a UI and use the overall data-flow of a process model including an analysis of all used data types to provide a set of valid change operations. For extended changes the additional question which kind of change operations make sense if they are executed in a user interface (template) has to be answered. This limits the level of complexity of such kind of change operations, e.g., adding a new branch in a XOR control-flow block is easy in a process model editor but hardly impossible if done by directly editing a user interface (template). Therefore the extended UI-based change operations are limited to, adding new human activities or rearranging existing human activities according to the underlying process model. Table 6.2 summarizes and classifies the problems to solve for user interface changes including the major requirements to provide solutions for them.

Problem Classification	**Facilitate Options**	
Change Granularity	local	extended
Affected Area . . .		
. . . in the Process Model	single activity, single control-flow block	single control-flow block including nested blocks
. . . in the UI Model (Screen)	single FormElementGroup, single FormTabTemplate	multiple FormTabTemplates
Supported Operations	Edit, New	Edit (Move & Rearrange), New
Guidance Generation	based on process model data-flow and data types	based on process model control-flow

Table 6.2: Classification of Problems to Provide User Interface-based Changes

Limiting the user interface change operations to capabilities like the existing data-flow of the underlying process model leeds to strong restrictions of the possible changes. This especially affects the creation of new Input and Output fields in a user interface editor since new fields only can be created if they are available in the process model data-flow. A possible improvement for this restriction would be the suggestion of adding new data objects to the process model data flow if a user wants to create new input or output fields in a user interface (template). However, this would lead to the problem of how to supply this new data objects with values by using the existing process model and then in turn result in the requirement of changing the process model, e.g., by adding a new activity, which creates and supplies these new data values.

Table 6.3 summarizes the effects and required actions on a process model for changes made in a user interface (template) editor. It groups the changes based on User Interface Model elements since all UI Widgets based changes result in User Interface Model changes. With

Affected UI Widget	*Step Tab*	
UI Model Object	*FormTabTemplate*	
UI Change Operation	New	Edit
Effect on Process Model	Create new Activity	Rearrange underlying Activity or control-flow block

Affected UI Widget	*Form Group*	
UI Model Object	*FormElementGroup*	
UI Change Operation	New	Edit
Effect on Process Model	Create new, Activity which is in a parallel branch located in a surrounding parallel block of the actvity representing the current form	Update / Rearrange underlying Activity

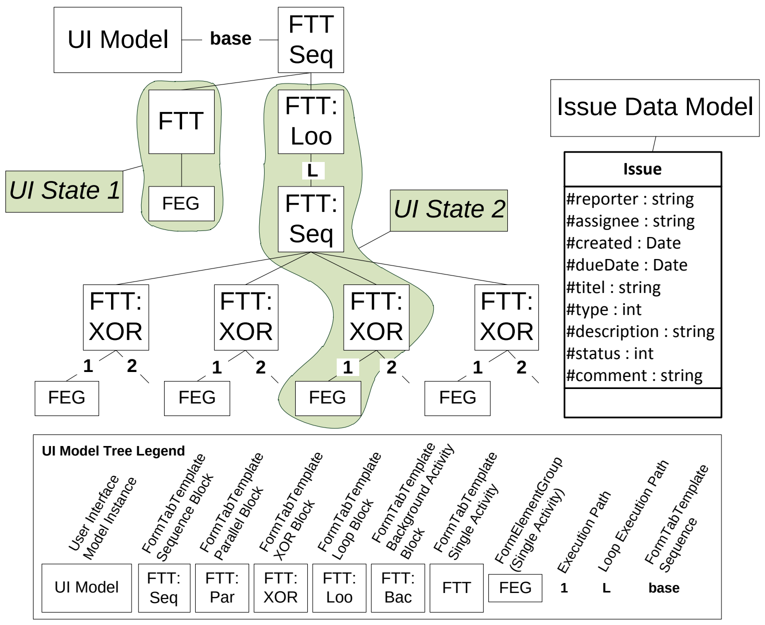
Affected UI Widget	*Input & Output Fields*	
UI Model Object	*FormElement*	
UI Change Operation	New	Edit
Effect on Process Model	Check if new FormElement can be supplied by the existing data-flow, if not create a service activity, as direct predecessor activity of the human activity linked to the affected form in the process model, which creates and supplies the data object	Update the respective data elements of the underlying activity

Table 6.3: Effects of User Interface-based Changes on the Process Model

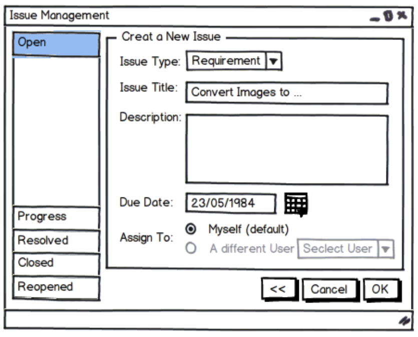
the help of these definitions, it is possible to proceed with the discussion of user interface changes made during the run-time of process instances. The run-time change related aspects are covered in Chapter 7. These run-time aspects include additional solutions for some of the problems outlined here. For instance, how to solve the missing data element provisioning and enable a UI for more advanced change operations by concerning declarative process modeling techniques [PvdA06]. Before starting with this the following chapter presents some results for generated user interfaces based on the use case process models introduce in Chapter 4.1.

6.2.5 Application of the Transformation Algorithm to the Use Cases

The User Interface Model presented in this chapter have been generated by the previously described Transformation Algorithm (cf. Chapter 6.2.3). Obviously, this also includes the usage of the Transformation Patterns (ETPs and CTPs). The presented UI Model trees represent all required elements until the level of the single activities for the respective process models. For simplicity reasons, the processed data elements have been modeled as UML class diagrams, which are included in the respective UI Model figures.



(a) Issue Management UI Model with UI States and Data Model for Role Administrator

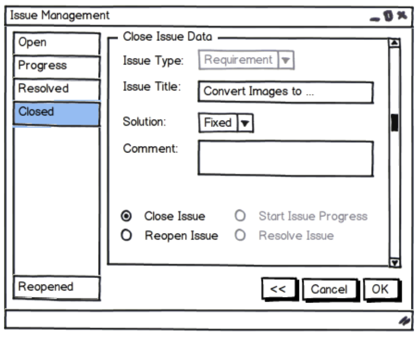

Issue Management

Open
Progress
Resolved
Closed
Reopened

Creat a New Issue
Issue Type: Requirement
Issue Title: Convert Images to ...
Description:
Due Date: 23/05/1984
Assign To: [✓ Myself (default)] A different User Seclect User

<< Cancel OK

(b) Issue Management, UI State 1


Issue Management

Open
Progress
Resolved
Closed
Reopened

Close Issue Data
Issue Type: Requirement
Issue Title: Convert Images to ...
Solution: Fixed
Comment:
[✓ Close Issue] Start Issue Progress
Reopen Issue Resolve Issue

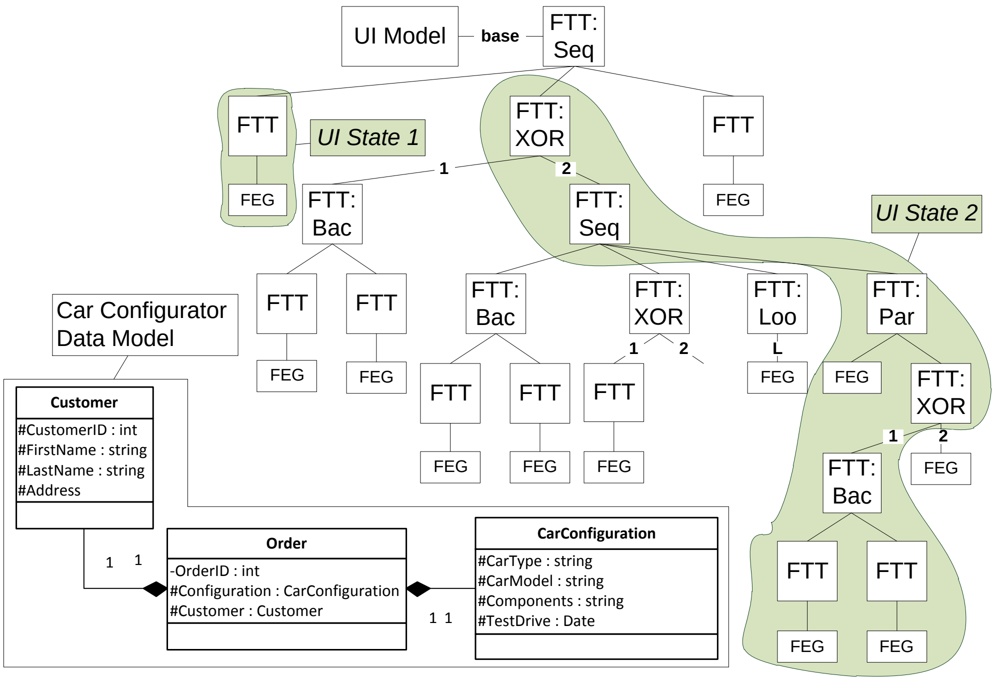
<< Cancel OK

(c) Issue Management, UI State 2

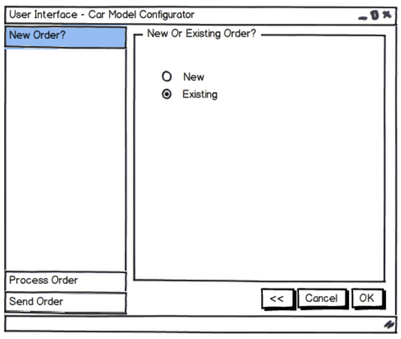
Figure 6.18: Use Case 1: Issue Management, Complex UI Result

Figure 6.18a shows the role-specific resulting UI Model in a tree-oriented notation, for the *Issue Management* use case (cf. Chapter 4.1.1) based on the *administrator* role. It includes a legend for the UI Model elements, which are used in the subsequent samples. Moreover, it includes the data model for an issue data object, which is the single data entity type processed by all activities of the underlying process model. In addition, two states in the UI Model are highlighted. Such a UI state refers to a certain state in the process model. More precisely, it illustrates the complex user interface capabilities by using the UI Model, to refer to the execution of certain activities or control-flow block elements of the process model.

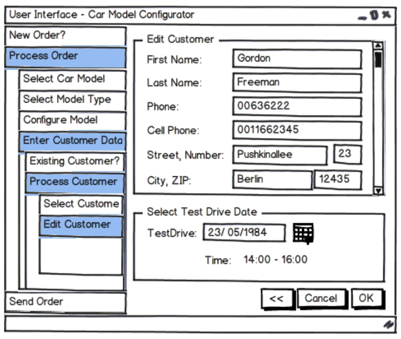
Figure 6.18b show the corresponding user interface for *UI State 1*. This UI state represents the activity *Create New Issue* in the underlying process model. Whereas Figure 6.18c shows the user interface for *UI State 2* and thus represents the corresponding *Close Issue* activity in the process model. For each of the XOR control-flow blocks both execution paths are included in the UI Model whereupon the empty path is always marked with **2**. Choosing this path at execution time would always result in directly skipping over to the next step.



(a) Car Configurator UI Model with Sample States



New Existing [✓]



Edit Customer

First Name: Gordon
Last Name: Freeman
Phone: 00636222
Cell Phone: 0011662345
Street, Number: Pushkinallee 23
City, ZIP: Berlin 12435

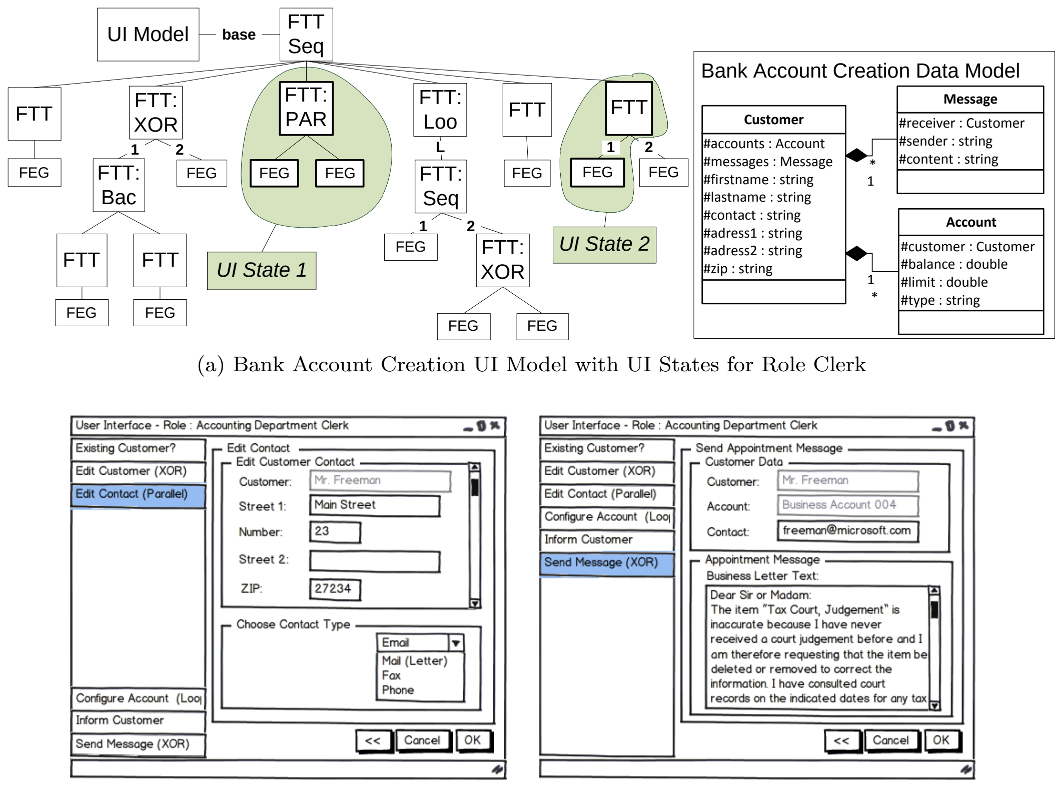
Select Test Drive Date

TestDrive: 23/ 05/1984
Time: 14:00 - 16:00

(b) Car Configurator, UI State 1 (c) Car Configurator, UI State 2

Figure 6.19: Use Case 2: Car Configurator, Complex UI Result

Figure 6.19 shows the resulting UI Model, for the *Car Configurator* use case introduced in Chapter 4.1.2. On the lower left side the corresponding data model is included as UML class diagram. It consist of, a configuration object that stores the individual configuration for a car model (CarConfiguration), the customer data (Customer) and the order that links a customer to an individual car configuration. The user interface for the highlighted *UI State 1* shown in Figure 6.19b represents the activity *New or Existing Order* in the process model. The user interface for *UI State 2*, shown in Figure 6.19c, cannot be mapped to a single activity. It represents the editing of an existing customer data set (CTP6 Background Activity) while choosing a test drive date (CTP2 Parallel Block) for the car to order.



Edit Contact
Edit Customer Contact
Customer: Mr. Freeman
Street 1: Main Street
Number: 23
Street 2:
ZIP: 27234
Choose Contact Type
Email
Mail (Letter)
Fax
Phone

Send Appointment Message
Customer Data
Customer: Mr. Freeman
Account: Business Account 004
Contact: freeman@microsoft.com
Appointment Message
Business Letter Text:
Dear Sir or Madam:
The item "Tax Court, Judgement" is inaccurate because I have never received a court judgement before and I am therefore requesting that the item be deleted or removed to correct the information. I have consulted court records on the indicated dates for any tax

(a) Bank Account Creation UI Model with UI States for Role Clerk

(b) Bank Account Creation, UI State 1

(c) Bank Account Creation, UI State 2

Figure 6.20: Use Case 3: Bank Account Creation, Complex UI Result

Figure 6.20a shows the role-specific resulting UI Model, for the *Bank Account Creation* use case introduced in Chapter 4.1.3. For simplicity reasons only the UI Model for the role *Clerk* is shown. For the role *Head of Accounting* a separate UI Model has to be generated. The processed data (UML class diagram on the right side) is based on the entities Customer, Message and Account. At which multiple Messages and Accounts can be linked to on customer data set. For the highlighted *UI State 1* Figure 6.20b shows the resulting user interface. It represents the parallel control-flow block in which customer data is edited and the type of contact for customer communication is chosen. Figure 6.20c shows the user interface in the *UI State 2*. This represents the activity *Send appointment Message* in last XOR block of the process model. In it, a message that informs the customer about the conditions of its newly created account is written by the clerk and send to the customer.

By a detailed introduction of the grouping aspects implemented in the Transformation Model and the description of how to integrate role specific UI capabilities by referencing an organizational model, further complex user interface requirements (cf. Chapter 4.2) are fulfilled. More precisely the requirement of generating role specific user interfaces. Moreover, the discussed advance grouping aspects handle problems like different process model granularity and enable the individual allocation of activities to certain user interfaces in a flexible manner.

The presented Mapping Meta Model builds the base for the process model to user interface Transformation Algorithm. Thereby it is essential to implement the requirement of keeping the references between process model and user interface elements. The introduced Transformation Algorithm uses the single components of the Transformation Model, namely the Transformation Patterns and the grouping aspects, to describe a generic overall process model to user interface transformation (cf. Chapter 6.2.3). Its generic nature is based on the usage of the Mapping Meta Model (cf. Chapter 6.2.1) and the User Interface Model (cf. Chapter 3.2.2), which decuples the algorithm form any implementation technology specific concerns. Through the introduction of the Representation component, a mechanism for concrete user interface generation based on a UI Model has been described (cf. Chapter 6.2.4). With help of a Representation and the usage of the Mapping Meta Model the propagation of changes from a user interface to a process model and vice versa were introduced.

The next chapter handles additional concerns for complex user interfaces according to process run-time with special respect to UI-based modifications and to retrospective changes of the process control-flow.

7 Runtime Aspects

In this chapter user interface aspects according to process run-time are discussed. As outlined in the Chapter 3.1.2, process run-time is defined as the time in which a business process model is instantiated and the involved resources execute the generated instances. With the previously described complex user interfaces, we want to enable advanced operations according to process control-flow navigation. In addition, a user should have the possibility to perform change operations to a user interface, e.g., add a new input field to a form. To provide these capabilities in the following Chapter 7.1 a more detailed definition about the involved artifacts and their interactions, which are necessary for user interface generation and change operations on user interface elements, is given. These basic definitions are followed by an approach which defines how to handle user interface element modifications in Chapter 7.2. In Chapter 7.3 the problems according to the retroactive modification of process control-flow and the thereby resulting implications for user interfaces are discussed. Finally Chapter 7.4 lists the problems which have to be solved for a proper implementation of the discussed run-time aspects.

7.1 User Interface Generation Compendium

Table 7.1 shows the relation between user interface and process elements in the modelling and run-time phase of a business process. As mentioned before, in the course of BPM process modelling is the phase in which a process model is defined. Process run-time is the phase in which instances of the previously defined process model are created and executed [Wes07].

As outlined by the Mapping Meta Model and the definitions according to change propagation in Chapter 6.2, it is possible to propagate changes from a user interface template, with the help of the User Interface Model, to the underlying process model. Thus, a user has the possibility to individually rearrange user interface elements, e.g., for usability reasons, without losing the allocation between those UI elements and the process model. In addition,

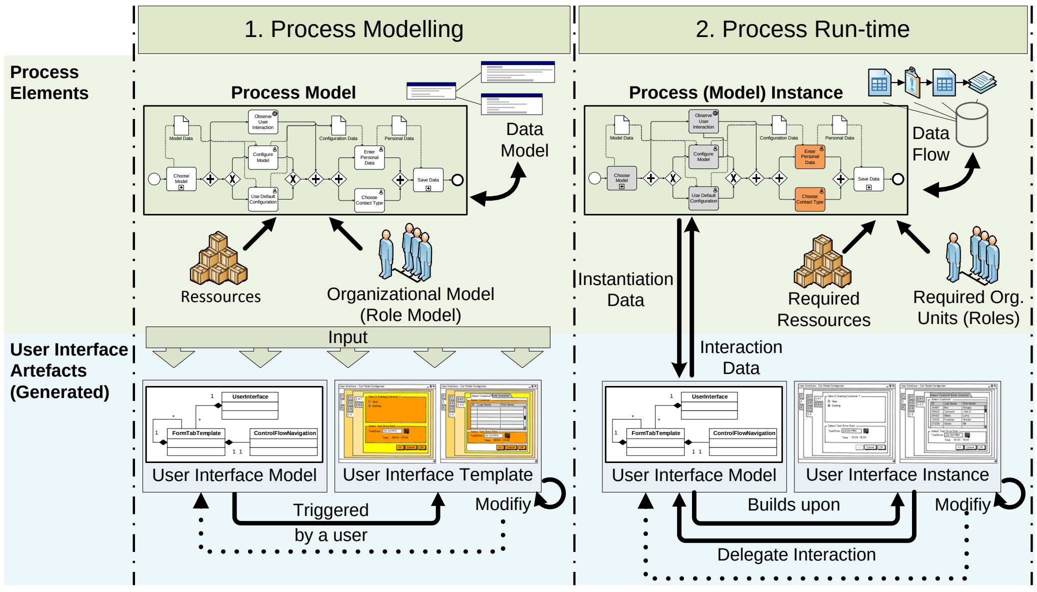


Table 7.1: User Interface Generation in the Process Life Cycle

it is possible to add new user interface elements and handle the implications, like adding new data input to an activity, in the process model like discussed in Chapter 6.2.4. These aspects are summarized by the left-handed *Process Modelling* column of Table 7.1 and are the base for the run-time definitions in the subsequent chapters. The right sided *Process Run-time* column of Table 7.1 gives an overview how user interface actions during process run-time are delegated to a process instance. The process instance or an instantiated process model is the base input to instantiate the User Interface Model with run-time data. The instantiated User Interface Model is then, in turn, used to generate a concrete user interface instance by implementing the previously introduced Representation interface. In general, the user interface instance is a user role-specific UI window, which includes all UI widgets need to perform the actions required while a process instance is executed.

During the execution of a process instance process run-time information, e.g., which execution path of a exclusive decision gateway has been chosen, is passed to the User Interface Model in a successive manner. This information can, in turn, be used to decide which UI elements to draw and which not. Since the execution progress of a process instance is directly related to the actions performed by a user in a respective user interface a bidirectional information exchange between the User Interface instance and the running process instance is necessary. Thus, the User Interface Model can been seen as an information broker for process- and corresponding user interface instances.

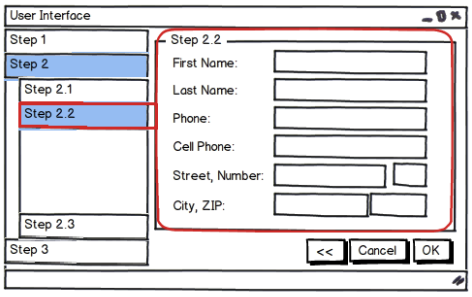
The broker nature of the User Interface Model during process run-time is also used to enable the UI instance for ad-hoc modifications like adding new input fields to a form. The details of these UI element modifications are described in the following chapters. In addition, the User Interface Model can support modifications to the process instance control-flow. Thus, a user can step back and pass through the control-flow in a different way up to a certain extent. These process control-flow rearrangements are discussed in Chapter 7.3.

7.2 UI Element Modifications

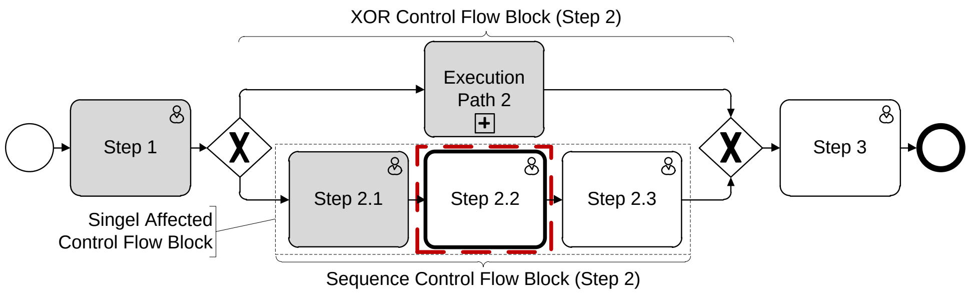
Element modification can be summarized as the action of adding new UI elements into an existing user interface during process run-time. The handling of these new UI elements, according to the underlying process model, differs significantly based on the type of UI element to add. Since different UI element types have a different change impact on the respective process instance. In the following this fact is called *change granularity*. Based on the distinction between two general change granularity levels, this chapter is divide into two subchapters. Chapter 7.2.1 describes possibilities how to deal with simple changes like adding a new input field to an existing UI element group. Chapter 7.2.2 which supplements the simple UI changes with the handling of more complex ones like adding a new FormTabTemplate element to a existing tab sequence, e.g., to express an additional execution option. The base of all run-time modifications presented in the following are the definitions of how to propagate changes as they have been defined in Chapter 6.2.4.

7.2.1 Basic UI Element Modifications

Basic UI element modifications are defined as changes made in the user interface instance during process run-time, which have no, or only slight change impacts on the process control-flow structure. It is equivalent with the local change granularity defined for propagating changes during process modelling. This covers all modification operations located on the actual visible UI screen. Based on the previously defined UI Model this refers to modifications, which are possible inside of a single FormTabTemplate element. Figure 7.1a shows the modification area of a user interface instance during process run-time. This modification area is a direct outcome of restricting changes. In addition, this results to restrict the impact of a UI-based changes only to certain control-flow blocks in the process model instance (cf. Figure 7.1b). For instance, if the FormTabTemplate element represents and parallel control-flow block and one of its branches contains a nested control-flow block a change can affect more than a single control-flow block.



(a) Modification Area based on UI Model FormTabTemplate element


XOR Control Flow Block (Step 2)
Execution Path 2
Step 1
Step 2.1
Step 2.2
Step 2.3
Step 3
Singel Affected Control Flow Block
Sequence Control Flow Block (Step 2)

(b) Affected Activity and Control Flow Block in the respective Process Instance

Figure 7.1: Process Instance Impact for Adding new Elements to a User Interface Instance

At first the performing of UI-based changes at run-time is similar as it has been described for modelling time (cf. Chapter 6.2.4). New UI Widgets can be added in a drag and drop manner by a user. This results in an update of the User Interface Model. These update operations are triggered by a Representation implementation. The Representation implementation creates the respective UI Model elements for the new UI Widgets. With the help of the Mapping Meta Model, the changes can be delegated to the respective process model parts.

The big difference is that changes to the user interface do not directly affect the underlying process model. Instead, they are propagated to the respective process instance and the corresponding used data model of this. Therefore, all checks for the compliance of user interface changes are based on their integration ability in the process instance.

Before performing the adding of new UI elements the question about which UI element is valid to be added in actual user interface state has to be answered. A first simple approach for this could be allowing all kind of basic UI elements to be added. This would cover all input and output form elements including group elements. This procedure can have a variety of effects according to the underlying process instance. First, if new arbitrarily output fields (labels that display data) are added to a user interface it has to be clarified where the data to display comes from. This in turn results in a case distinction with three options, which are listed below:

1. Data to display is provided by previous activity.
2. Data to display is included in the currently used input data objects.
3. Data to display requires a new activity, which creates the necessary data elements.

For the first of these possibilities, the activities, which are connected by process control- or data-flow, with the respective activity of the modified user interface elements, have to be observed according to their used data elements. If one of these data elements matches the newly output field to add, e.g., based on the data type, it can be used as data input. The second possibility also requires a data observation but for this, it is limited to the input data objects of the activity. If the first two options are not successful, the third option can be used as an escalation strategy. For this a new service activity which is directly included before the actual activity has to be created. This new service activity has to provide the required data elements and send them to the activity responsible for the current user interface. This, in turn, leads to the fundamental question where from this ad-hoc create service activity should take the required data. An example solution for this could be the definition of some kind of data source for process models whose instances should be enabled for UI-based changes. More details of this data providing problem are outlined in the Chapter 7.4.

To add new input fields to a user interface the thing that has to be clarified is where do the new input fields send their data to? As for the output fields this results in a distinction of three cases, which are listed below:

1. Input data is provided for a subsequent activity.
2. Input data is included in the currently used output data objects.
3. Input data is send to a new activity, which processes the new data elements.

To use the first option a definition of what to do with the data in later process steps is required. By default an activity, which already processes data, from the underlying activity of the modified UI screen, can be used as input for the new data values. This leads directly to the second option, which can be seen as a special case of the first option. It requires the activity to output business data objects based on the properties of this business data objects the values of the new added input data elements can be attached to a matching business object, again based on an analysis by matching the data types of the business object properties against the underlying data types of the new input fields.

Another option is the creation of a new service activity directly after the user interfaces activity, which consumes the data values of the new input data elements. Of course, these attempts do only make sense if the correct processing of the new data elements is assured, e.g., an underlying data base system has the capabilities to store the new data values. In addition to the read and write UI elements it is possible to add UI elements to edit data. To achieve this correctly according to the process instance in use both criteria the ones for read data elements and the ones for write data elements have to be fulfilled.

Due to the fact that in our UI Model approach user interface group elements are used to group the UI elements based on the data-flow of a single activity and to group the primitive data elements of a business object a distinction if adding a new group element to a UI is required. If adding a new group element to a UI the desired behaviour has to be specified. Valid options are the creation of a new activity, which is arranged parallel to the current activity, or to supplement the current activity with a new business object to process. Table 7.2 summarizes the required actions on a process instance based on the kind of UI element to add to a user interface.

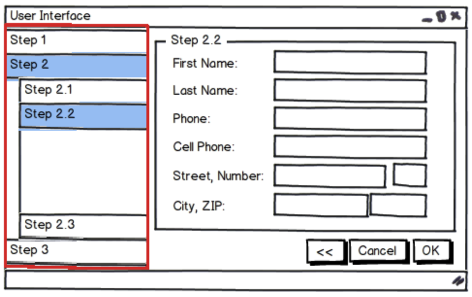
Kind of UI Element	**New User Interface Element to Add**		
	Input	***Output***	***Group***
New required Process Element	**Effects on Process Model Instance**		
	Data Output (by previous Activity)	Data Input (by subsequent Activity)	New Activity (Arranged parallel to current Activity)
			Data Ouput or -Input based on a Business Object
Where does the new Data ...	**Options to achive correct Data Supply**		
... come from?	n/a	Previous Activity	Previous Activity
		Activity Data-Flow	Activity Data-Flow
		New Activity	New Activity
... go to?	Subsequent Activity	n/a	Subsequent Activity
	Activities Data-Flow		Activities Data-Flow
	New Activity		New Activity

Table 7.2: Effects on Process Model Instance by User Interface Element Modifications

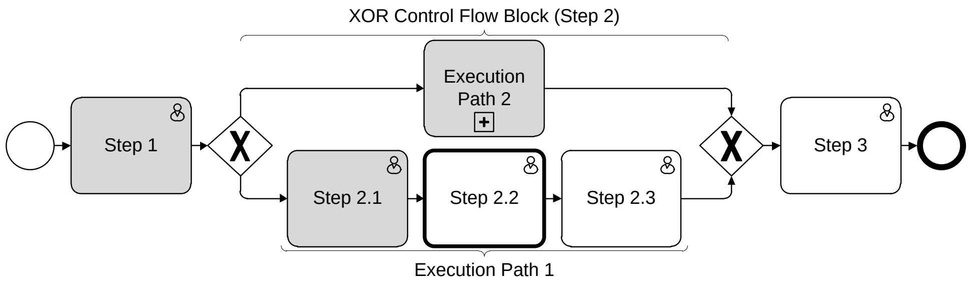
To assure the correct execution of the running process instance it makes sense to restrict the change operations. One course of action to achieve this would be an analysis of the data model used by the process instance and the control-flow connections of the activity representing the current user interface. Based on the results of this analysis a catalog of possible valid changes to the user interface could be generated. More details about this change catalog generation are presented in Chapter 7.4. In the following, the required actions for more complex UI modifications, like adding complete new tab elements to a user interface, are discussed.

7.2.2 Advanced UI Modifications

In the following, we will outline the properties of advanced modification in user interfaces during process run-time. The advanced modifications differ from the previously introduced basic modification because these modification cause changes of the process control flow structure. This control flow modification is only limited up to a certain extend. The limitation factor is the practicability of performing these changes in a user interface.



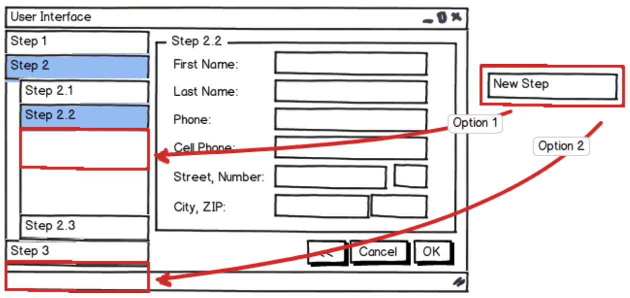
(a) Advanced Modification Area Located in the Process Instance Progress Menu of the UI



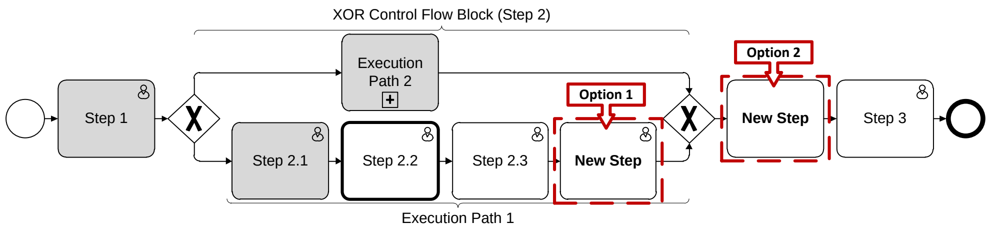
(b) Process Instance Including the Actual State for User Interface

Figure 7.2: Advanced Modification Area and Respective Process Instance

Figure 7.2b shows the origin for the advanced UI-based modifications in a complex user interface. Expect of the basic UI modifications these are located on the left sided process instance progress menu. Located in this process instance progress menu the operations add, rearrange (move) and remove (delete) are supported. An important prerequisite before performing an advance change from within a user interface instance is to check if the change is possible according to the progress of the underlying process instances. A simple example for an invalid modification operation would be the adding of a new step (activity) behind the already processed activities. Figure 7.2b shows the respective minimal example process instance for the UI of Figure 7.2b, including the actual progress state (the activity of Step 2.2 is activated). For this purpose, the already finished activities are highlighted in grey color the actual activated activity is marked by a thick border. In the following, this minimal process instance example is used to discuss the different modification operations.



(a) Two Valid Options for Adding a new *Step* within a User Interface



(b) Two Insertion Options for Resulting new Activities in the Respective Process

Figure 7.3: UI-based Add Operation during Process Run-time

As mentioned before the adding of a new step within a user interface during the run-time of a process instance requires that the corresponding activity to add in the process model be after the actual activated activity. Figure 7.3a shows two valid options for adding a new additional step in sequence to existing ones within a user interface. For this, Figure 7.3b illustrates the resulting options for adding the respective new activities in the process model.

In *Option 1*, the activity is added to the actual executed path (*Execution Path 1*) of the XOR control-flow block. Whereas in *Option 2* the activity is added to the global sequence control-flow block, between the XOR block and the last activity (*Step 3*). Based on the block-oriented process model structure and the therefore resulting complex user interface structure, the only limitations for adding new steps in a UI are:

1. The new step has to be after the actual execution position based on overall process model control-flow.

2. Dead execution path like *Execution Path 2* are restricted from adding new Elements. Whereas in the example of Figure 7.3 this is impossible anyway since after *Execution Path 1* has been chosen the complex UI only shows the steps necessary for this execution path. Thus, all UI-based changes, which are made beside the hierarchy level of *Step 2* result in changes in the correct execution path.

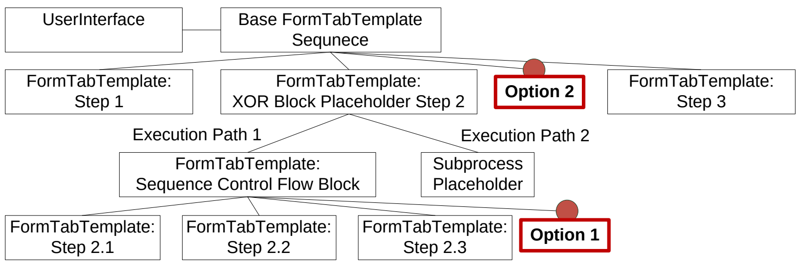
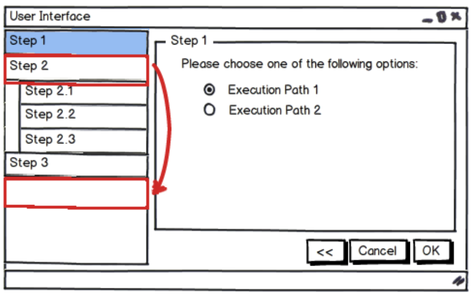

UserInterface
Base FormTabTemplate Sequnece
FormTabTemplate: Step 1
FormTabTemplate: XOR Block Placeholder Step 2
Option 2
FormTabTemplate: Step 3
Execution Path 1
Execution Path 2
FormTabTemplate: Sequence Control Flow Block
Subprocess Placeholder
FormTabTemplate: Step 2.1
FormTabTemplate: Step 2.2
FormTabTemplate: Step 2.3
Option 1

Figure 7.4: Affected parts in a UI Model Instance for Add Options

Referring to our UI Model, the execution of the suggested add operations will always result in the creation of a new FormTabTemplate element, which contains a single FormElementGroup. The difference between Option 1 and 2 is the hook in point for this new element. For Option 1 this hook in point would be the FormTabTemplate of the sequence control-flow block contained in the first execution path of the XOR block. Whereas for Option 2 this would be the base FormTabTemplate set contained in the UI Model (cf. Figure 7.4).

However, adding this new steps during the run-time of a process is only one thing to do if implementing such a user interface based change system. An important question to answer is what is the semantic of this new activity and what data elements are required for the desired processing of these new steps and in addition where should read data elements come from and written data elements go to, to enable a reasonable finishing of the underlying process instance. For this purpose solutions like they have been described in Chapter 7.2.1 to realize basic element modifications, e.g., analyzing the overall data flow of the process model an suggesting respective data elements for the processing in the newly added steps, could be a basic approach.

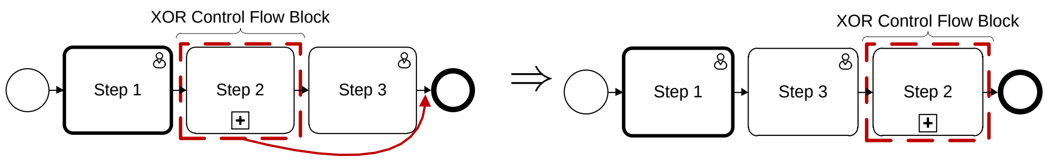

User Interface

Step 1
Step 2
Step 2.1
Step 2.2
Step 2.3
Step 3

Step 1
Please choose one of the following options:
[✓ Execution Path 1]
Execution Path 2

<< | Cancel | OK

(a) Moving a *Step* within a User Interface



(b) Process Instance before and after the User Interface based Move Operation

Figure 7.5: UI-based Move Operation during Process Run-time

The next possible modification operation is the moving of existing elements in the user interface. Figure 7.5a illustrates such UI-based moving operation. In this example, the user interface element *Step 2* is moved behind *Step 3*. This includes the moving of all child step elements contained in *Step 2* (*Step 2.1* to *Step 2.3*) and results in a change of the execution order. Based on this Figure 7.5b visualizes the change operations necessary in the underlying process model. The XOR control-flow block (which is shown in a collapsed form for simplicity reasons) has to be moved from its source position behind the activity for *Step 3*.

In this example, the effected process model part is a complete control-flow block. The move operation during process run-time shares the prerequisites according to the state of the underlying process model instance with the previously described add operation. Thus, it is only possible to move parts in the UI, which have not been executed to other unexecuted parts. Additional restriction according to move operations are the control-flow block nesting level and the data elements which are processed by the moved activities. Move operations should for simplicity reasons always be based in the same block nesting level deep. A more important restriction aspect are the data elements processed by activities affected by an

UI-based move operation. If an activity provides data elements for a subsequent activity and the data providing activity would be moved behind the activity requiring the data, the process instance would be stocked in a deadlock since the required data could never arrive.

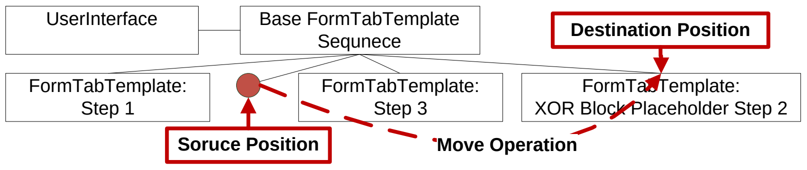
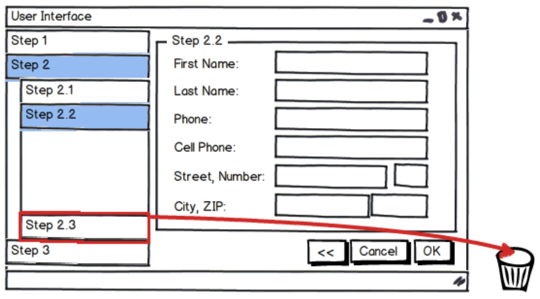
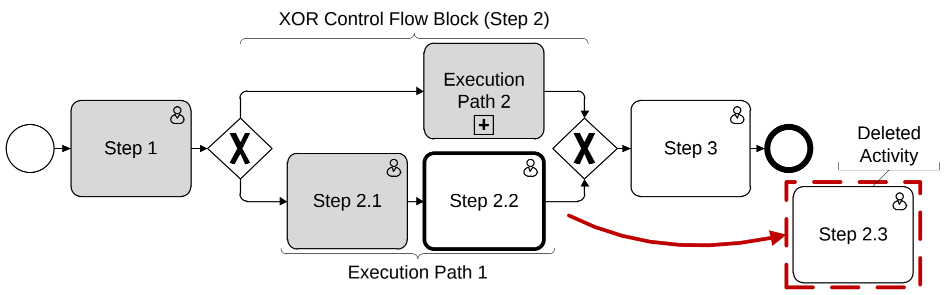


Figure 7.6: Advanced Move Operation in a UI Model Instance

Figure 7.6 visualizes the UI Model changes necessary for the UI-based move operation example of Figure 7.5. The implementation of this is rather simple since only the processing order of the FormTabTemplate elments in the base FormTabTemplate sequence of the UI Model instance has to be changed.



(a) Deleting a *Step* within a User Interface



(b) Deletion of Corresponding Activity in the Respective Process Instance

Figure 7.7: User Interface based Deletion of an Activity

The last presented advanced modification operation is the deletion of complete steps within the user interface. For such a delete operation Figure 7.7a shows an example. In this example, the left sided tab selection UI Widget of *Step 2.3* is deleted. This results in the deletion of the complete underling form elements and, in turn, in the deletion of the respective activity in the running process instance. Therefore, Figure 7.7b illustrates the deletion of the respective activity *Step 2.3* in the running process instance. It is just removed from its position in the corresponding sequence control-flow block.

The prerequisites to allow such a deletion operation are quite similar as for the previously presented move and add operation. Deletion of steps and, in turn, their underlying activities is only possible if the activities are prior to the actual execution position of the process instance according to the overall process control-flow. Additionally it should be mentioned that the deletion of already executed steps is in some way pointless. Another restriction is based on the overall data-flow. If an activity is responsible to provide data for subsequent activities, the deletion of the respective user interface step element is not possible since this will result in a blocked process instance. The reason for this is the fact that by the deletion of the step element the respective input form UI Widget that was responsible to create the later required data is completely missing.

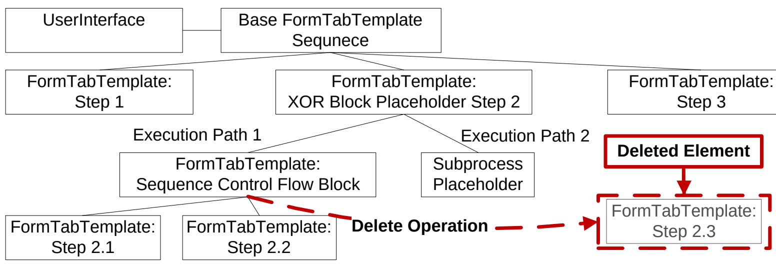

UserInterface
Base FormTabTemplate Sequnece
FormTabTemplate: Step 1
FormTabTemplate: XOR Block Placeholder Step 2
FormTabTemplate: Step 3
Execution Path 1
Execution Path 2
Deleted Element
FormTabTemplate: Sequence Control Flow Block
Subprocess Placeholder
FormTabTemplate: Step 2.1
FormTabTemplate: Step 2.2
Delete Operation
FormTabTemplate: Step 2.3

Figure 7.8: Advanced Delete Operation in an UI Model Instance

Figure 7.8 illustrates the impact of the delete operation on the UI Model instance based on the delete operation sample of Figure 7.7. The only required change is the removal of the FormTabTemplate element associated with *Step 2.3*. Since this FormTabTemplate is located in a leaf of the tree like data structure, this has no impact on different UI Model parts.

Summing up advanced UI-based change operations (add, move and delete) during the runtime of a process instance require a detailed analysis of the further data processing. In addition the actual execution state of the process instance limits the performance of UI-based change operations to process elements (control-flow blocks & activities). Thus, it is only possible to perform changes on UI elements which are associated with process elements succeeding the actual activated activity according to the overall process control-flow.

7.3 Sequence Modifications

In this chapter the control-flow navigation options of complex user interfaces are discussed. These navigation options are summarized under the term *Sequence Modifications*, since they enable a user to modify the processing sequence of already executed process instance parts. The navigation options, which are provided in a complex user interface have and direct associated counterpart in the UI Model. During process run-time, theses control-flow navigation parts of the UI Model are used to enable or disable the respective actions in the actual user interface.

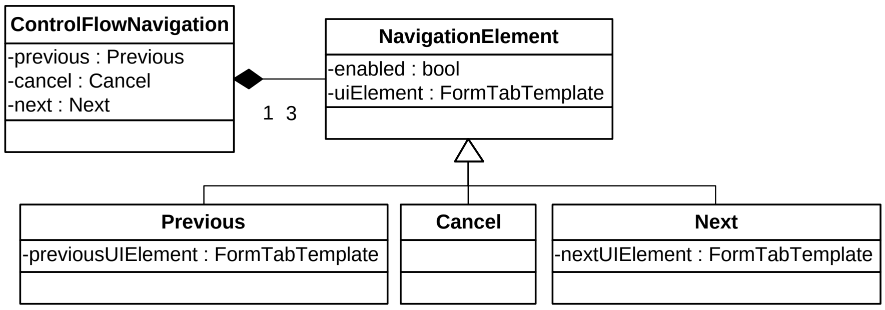


Figure 7.9: UI Model ControlFlowNavigation UML Class Diagram

Figure 7.9 shows a UML class diagram for a *ControlFlowNavigation* element. In the UI Model such a ControlFlowNavigation instance is associated with each FormTabTemplate element. This enables the navigation between different single FormTabTemplate elements. The initialization of the single parts (Previous, Cancel & Next) of this ControlFlowNavigation is based on the underlying process model respectively the process instance control-flow. It is realized with help of the previously introduced Mapping Meta Model (cf. Chapter 6.2.1) that enables the passing of the required information, e.g., which is the next activity, between the process instance and the UI Model. The ControlFlowNavigation can be seen as the link structure between single entry elements like in a doubly linked list. In the following, the three actions enabled in a complex user interface by this are presented:

1. The **Next Step** action is realized as OK button. If this button is pressed by a user, the form elements shown in the actual user interface screen are processed as defined by the underlying process model. In general, this is the execution of the underlying human activities. Following this, the next UI screen elements represented by a FormTabTemplate of the UI Model are displayed in the complex user interface and if necessary the left sided process instance progress menu is updated.

2. The **Clear Data** action is realized by the Cancel button. It just represents a convenient operation, which clears the data of all input form elements in the currently displayed UI screen.

3. The **Previous Step** action is realized as Back button. If this button is pressed, the predecessor UI Model elements according to process model control-flow are loaded. By default, the associated UI Widgets of this already processed elements are only displayed all modification options are disabled, since the data elements shown and thus the respective underlying process instance activities have already been executed. In some special cases, the enabling of already executed activities is possible. This rather complex circumstance named Sequence Modifications is discussed in the following.

An example for such a Sequence Modification would be the re-execution of an already processed XOR gateway by stepping back to the user interface form in which the decision for a specific execution path of this XOR gateway has been triggered. If this decision was based on input data made in a user interface form then it would be possible to choose a different option and thus a different execution path as in the first time executing the form.

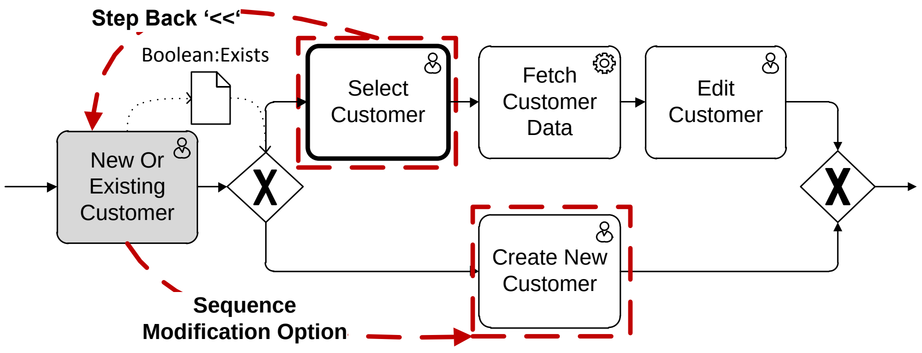
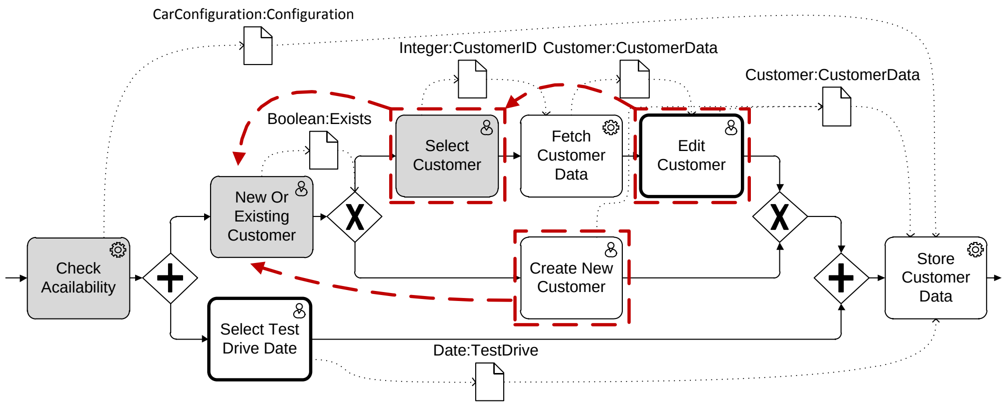
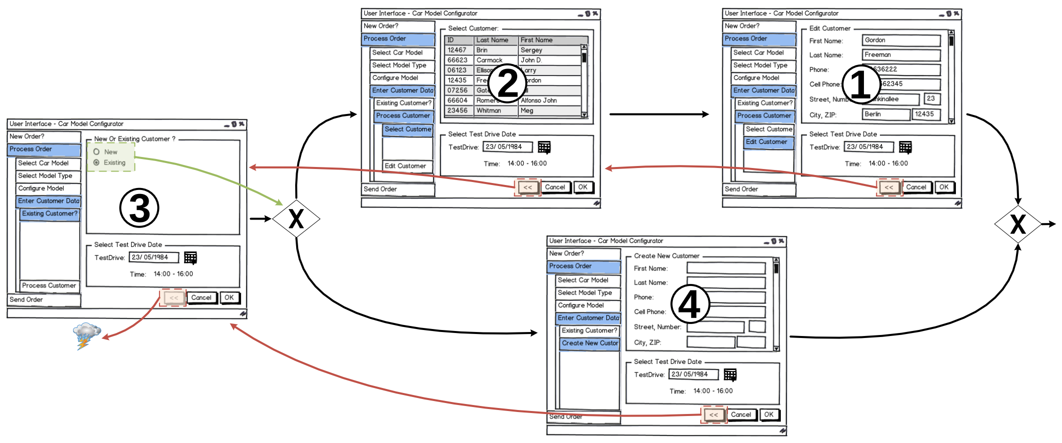


Figure 7.10: Process Model Snippet for Sequence Modification Example

Figure 7.10 shows a sample process instance snippet (taken from the Car Configurator use case). In this *Select Customer* is the actual activated activity. In the respective user interface for this activity it would be possible to step back and re-execute the *New Or Existing Customer* activity and choose the option to create a new customer instead of editing an existing one. The reason for this that in the *Select Customer* activity no process instance relevant data elements have been modified. This circumstance can be generalized to a prediction mechanism whether it is possible to re-execute certain activities or not. Which, in turn, results in a mechanism for enabling or disabling form elements in a complex user interface if the Back button is used.



(a) Car Configurator Process Instance including Processed Data Elements



(b) Resulting Complex User Interface Instance Flow

Figure 7.11: Sequence Modification Example: Car Configurator

This requires an analysis of all already processed activities according to their processed data elements along the original chosen execution path in the process instance. In a basic implementation the decision whether a human activity could be re-executed or not could be based on the criteria if this human activity includes write data elements which are processed by subsequent service activities or not. More general the distinction between enabling or disabling human activity re-execution is based on the fact if previous activities have generated relevant data for the later process execution.

Figure 7.11 shows a more complex example for the calculation of such step back paths in a process instance. It uses the Car Configurator process enriched with more details like service activities and the data-flow (cf. Chapter 4.1.2). In Figure 7.11a the activated activates are *Edit Customer* and to this parallel activity *Select Test Drive Date*. The corresponding user interface instance for this state is labeled with **1** in Figure 7.11b. Starting from this situation it is now possible to step back two times (user interface instances **2** & **3**) since no data elements for the later resuming of the process instance has been generated. In the state **3** of the sample UIs a further step back is not possible, but the decision made in this step can be re-executed with a different choice as before. This results finally in the UI state **4** (activation of the *Create New Customer* activity).

Concluding this *step back path calculation* is a possibility for simple, UI-based control-flow sequence modification based on the analysis of the data element types processed by already executed activities.

7.4 Discussion

In this chapter, the previously presented concepts of user interface based changes during process run-time and the modification options for process control-flow are evaluated. The basis for both concepts are the data elements in use by the process instance activities. According to element modification and in particular to the case of adding new UI Widgets to a user interface the general question to answer is: where to get required data and where to put newly created data to. A general answer for this question is missing. In particular, this problem can be solved by using data elements, which are already contained in the process instance at least to display it (new UI Model output elements). Alternatively, by adding service activities, which process the data of newly create input elements. The problem in this case is what exactly the new service activity should do with the received data elements. A general ad-hoc solution for this seems to be impossible. Since on the one hand this requires technical knowledge how to configure data sources and on the other hand makes the UI-based change operation to difficult for a regular user.

An option for a partial solution of this problem could be the predefinition of certain domain-specific changes already within the process model, including potential insertion points. This would result in a process model which contains additional declarative process model parts, e.g., as they have been described in [PvdA06]. If during process run-time a user request a change based in the UI a list of actual valid change operations could be presented. An example for this would be a medical examination, which contains a small set of standard steps. In which during the examination occurs, additional required steps could be added from a list of predefined steps.

For user interface-based delete and move operations the situation is similar. The data-flow related with the respective activities is essential. Move operations are only possible if the overall data supply of the actual process instance still can be fulfilled. Based on the fact that input forms often serve as data provider for the directly following activities UI-based move operations are only possible to a limited extend. For instance, as shown in the example case if complete process parts are moved. For delete operations the situation is quiet similar. In addition, the deletion of input form-based UIs will result in more than just the deletion of the single underlying human activity. Moreover respective service activities which are connected with the human activity, e.g., for processing the input form data, have to be removed from the process instance. A complete discussion of this process instance change related aspects can be found in [WRR07] and should not be included here in all its details. [WRR07] also discusses the migration from process instance-specific changes to the underlying process model. This aspect of migrating process instance-specific changes to the underlying process model is another aspect which could be included in UI-based modifications.

The based for the described sequence modification is calculating possible return paths from within the actual execution state of a process instance. This calculation is based on processed data elements. A return in the control-flow is only possible if these data elements are not relevant for the overall execution of the process instance. If such a step back is performed the underlying process instance has to be modified respectively, e.g., already executed activities have to be marked as unexecuted again. A simple situation in which such a return is impossible is for example the execution of an input form to create a customer data set. If the form has already been processed and the following activity is activated the situation for a possible step back is not clear. Since resetting the customer creation activity might result in the deletion of the underlying customer data set. However if the deletion of data elements is technically not supported this is just impossible. In such a case, it is possible to just disable the input fields of the form and display the data the user has entered before without performing any modifications according to the underlying process instance.

Referring one more time to the complex user interface requirements (cf. Chapter 4.2) defined based on the introduced use cases (cf. Chapter 4.1), with the help of the described Sequence

Modifications all of theses requirements are covered. Furthermore, the process run-time based user interface modification operations are beyond these initial defined requirements. Thus, in the next chapter a prototypical implementation for the Transformation Model, covering the most important of these requirements is presented.

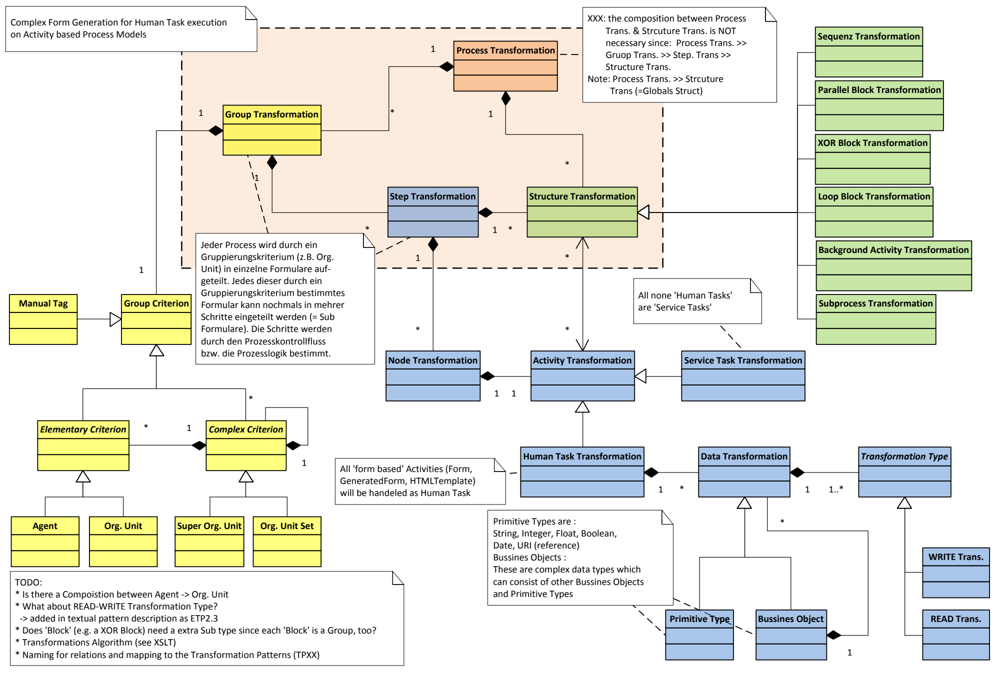
8 Prototypical Implementation

The developed prototype presented serves as proof of concept for the core aspects of complex user interface generation as already described. In particular, it implements the Transformation Patterns (ETPs and CTPs), a simplified Activity Grouping Mechanism and the Transformation Algorithm including all essential required parts. The UI generation prototype has been implemented as part of the *proView* [RKBB12] research project, which tries to handle process model complexity by the generation of individual simplified process model views.

In Chapter 8.1 the methodology according to the performed development is presented. This is followed by Chapter 8.2, which handles mainly the implementation technique related concerns of the prototype, supplemented with problems and their solutions occurred during the development. Finally, Chapter 8.3 discusses the results and outlines some improvement proposals.

8.1 Methodology

The starting point for the prototype implementation was on the one hand side the definition of use cases (cf. Chapter 4.1) and on the other hand side a detailed analysis of the ADEPT2 [DRRM$^+$10] process model, since the *proView* [RKBB12] prototype uses ADEPT2 process models for its view generation. The storage format for ADEPT2 process models is XML-based. Thus XSLT has been used for first transformation experiments [W3C07]. The results for these experiments showed up positive, in the sense that it was possible to extract the required information about activities (nodes) and the corresponding data-flow. The results of these experiments have been used as input for the definition of the previously introduced Transformation Patterns. Based on the Transformation Patterns and the Activity Grouping aspect a first design draft of the prototype has been created.



Complex Form Generation for Human Task execution on Activity based Process Models

XXX: the composition between Process Trans. & Strcuture Trans. is NOT necessary since: Process Trans. >> Gruop Trans. >> Step. Trans >> Structure Trans.
Note: Process Trans. >> Strcuture Trans (=Globals Struct)

Jeder Process wird durch ein Gruppierungskriterium (z.B. Org. Unit) in einzelne Formulare auf-geteilt. Jedes dieser durch ein Gruppierungskriterium bestimmtes Formular kann nochmals in mehrer Schritte eingeteilt werden (= Sub Formulare). Die Schritte werden durch den Prozesskontrollfluss bzw. die Prozesslogik bestimmt.

All none 'Human Tasks' are 'Service Tasks'

All 'form based' Activities (Form, GeneratedForm, HTMLTemplate) will be handeled as Human Task

Primitive Types are :
String, Integer, Float, Boolean, Date, URI (reference)
Bussines Objects :
These are complex data types which can consist of other Bussines Objects and Primitive Types

TODO:
* Is there a Compoistion between Agent -> Org. Unit
* What about READ-WRITE Transformation Type?
  -> added in textual pattern description as ETP2.3
* Does 'Block' (e.g. a XOR Block) need a extra Sub type since each 'Block' is a Group, too?
* Transformations Algorithm (see XSLT)
* Naming for relations and mapping to the Transformation Patterns (TPXX)

Figure 8.1: Conceptual Data Model for first Iteration Transformation Algorithm Prototype

Figure 8.1 shows the conceptual data model for this prototype. In it, the components for creating a Representation and integrating a real process model has not been defined. However, this *Transformation Data structure* was used for the first implementation of the Transformation Algorithm. Together with a User Interface Model implementation and based on manual tests with the use cases, the process model to user interface Transformation Algorithm has been validated. After this first proof of concept, integration tests according to the usage of the proView prototype according to its process view generation capabilities were performed. In addition, the vaadin[1] MVC web framework has been analyzed according to its UI generation capabilities, since it is already in use in the proView prototype. Based on the results of the Transformation Algorithm prototype and the proView integration tests a design for a complex UI generation prototype was developed and implemented. The design and implementation approach of this complex UI generation prototype is described in the next chapter.

[1]`https://vaadin.com/`, last checked January 30, 2012

8.2 Implementation

The goal for the prototype implementation described in this chapter was to realize a complex user interface generation for block-oriented process models. Since this prototype should serve as an prove of concept advanced concepts, like the enabling of UI-based changes and the connection of the prototype to a BPMS for using the generated complex UIs to execute process instances, have been omitted. This results in the following list of required features which have to be implemented referring to the previously introduced concepts:

1. Implementations for both kind of **Transformation Patterns** are necessary, namely:
   a) **Elementary Transformation Patterns**, are necessary for generating the fields (labels and input fields) representing the single content elements of a complex user interface.
   b) **Complex Transformation Patterns**, to transform the control-flow block structure of the process model and with the help of this create a navigation menu and decide which content elements (created by ETPs) to display in the resulting complex UI.
2. **Activity Grouping** is used for the generation of role-specific user interfaces based on the roles described in the process models.
3. The **Transformation Algorithm** is implemented to provide the overall feature of complex UI generation.
4. The **User Interface Model** is partly implemented and used to separate the process model from user interface related technical implementation concerns.
5. The implemented **Representation** serves as controller, according to a MVC-oriented architecture, it handles UI-based interactions (including an initial creation of UI elements) and performs the resulting operations on the data model.

In the following, details about the single implementation parts and their intent according to a complex user interface generation are presented. Therefore Figure 8.2 shows an overview of the layered-based architecture of the prototype including a numbered label (**1.** to **3.**) for each of the layers on the left side. The first of this layers (**1.**) is build by the AristaFlow BPM Suite (cf. Chapter 2.2.1). It implements the ADEPT2 process model which is used as basic process model implementation in the UI generation prototype [DRRM^{+}10]. The

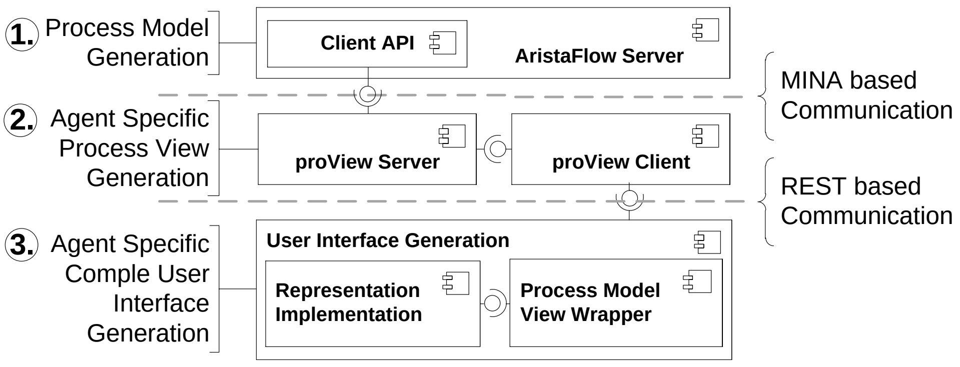

1. Process Model Generation
Client API
AristaFlow Server
MINA based Communication
2. Agent Specific Process View Generation
proView Server
proView Client
REST based Communication
3. Agent Specific Comple User Interface Generation
User Interface Generation
Representation Implementation
Process Model View Wrapper

Figure 8.2: Complex User Interface Generation Prototype Layer Architecture

AristaFlow Server provides a Java based Client API this is used by the *proView Server* in the second layer of the overall architecture (**2.**).

The proView Server provides a mechanism to generate views of a process models. A view of process model is a modified version of the original process model. Typically the modification operations for creating a process model view remove certain parts (nodes/activities) of the original process model [RKBB12]. Thus, the process view generation provide by the proView Server is exactly what is required to implement the Activity Grouping for complex UI generation. The view generation mechanism is used to create a view of a process model based on Grouping Criteria, e.g., the agents required for the performance of a certain process. The *proView Server* provides a lightweight REST-based interface for various different process view operations that is implemented by the *proView Client.* The proView Client, in turn, provides a Java-based API which is used in the *User Interface Generation* implementation (**3.**)

The User Interface Generation layer (**3.**) consists of two components. At first, the *Process View Wrapper* which implements the features of CTPs, Activity Grouping (by using the proView Client), the Transformation Algorithm and the User Interface Model. The second component is the *Representation Implementation* it realizes the concepts of representing a user interface as described in Chapter 6.2.4. Hence, the Representation is responsible for the generation of all user interface elements, their connection with the UI- and process model and the handling of events triggered by user interface elements (respectively user interactions with them). In the prototype this Representation has been implemented using the *vaadin* MVC web framework.

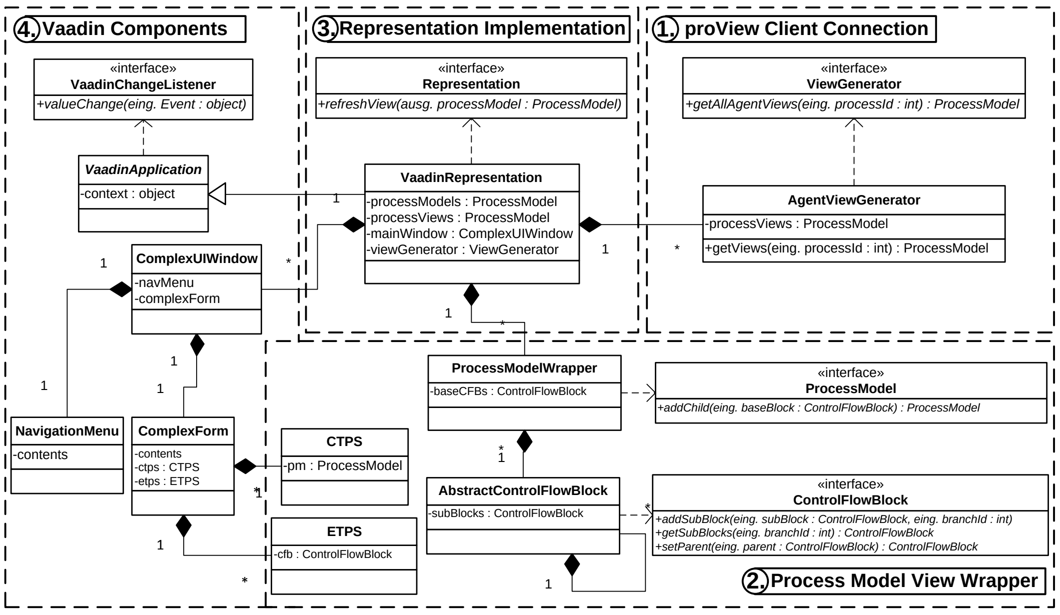

4. Vaadin Components
3. Representation Implementation
1. proView Client Connection
2. Process Model View Wrapper
«interface» VaadinChangeListener
+valueChange(eing. Event : object)
VaadinApplication
-context : object
«interface» Representation
+refreshView(ausg. processModel : ProcessModel)
VaadinRepresentation
-processModels : ProcessModel
-processViews : ProcessModel
-mainWindow : ComplexUIWindow
-viewGenerator : ViewGenerator
«interface» ViewGenerator
+getAllAgentViews(eing. processId : int) : ProcessModel
AgentViewGenerator
-processViews : ProcessModel
+getViews(eing. processId : int) : ProcessModel
ComplexUIWindow
-navMenu
-complexForm
NavigationMenu
-contents
ComplexForm
-contents
-ctps : CTPS
-etps : ETPS
CTPS
-pm : ProcessModel
ETPS
-cfb : ControlFlowBlock
ProcessModelWrapper
-baseCFBs : ControlFlowBlock
«interface» ProcessModel
+addChild(eing. baseBlock : ControlFlowBlock) : ProcessModel
AbstractControlFlowBlock
-subBlocks : ControlFlowBlock
«interface» ControlFlowBlock
+addSubBlock(eing. subBlock : ControlFlowBlock, eing. branchId : int)
+getSubBlocks(eing. branchId : int) : ControlFlowBlock
+setParent(eing. parent : ControlFlowBlock) : ControlFlowBlock

Figure 8.3: Vaadin-based Representation Implementation

Figure 8.3 illustrates the vaadin-based implementation of the Representation as UML class diagram. The *proView Client Connection Part* (**1.**) takes care for generating agent (based on the organizational model used by the underlying process model) specific process model views, by using the previously introduced proView Client API. This is a basic implementation of the Activity Grouping aspect. The *Process Model View Wrapper* (**2.**) part is used to transform the ADEPT2-specific process model view into an explicit hierarchical control-flow block based process model structure as described in Chapter 3.1.1. In addition, it contains the implementation of the ETPs and CTPs, which are combined in the also contained Transformation Algorithm implementation. Last but not least the *ProcessModelWrapper* class serves as a simplified version of UI Model.

The *Representation Implementation* (**3.**) part is the core part and serves as central controller for the whole prototype. Thus, it fetches the process model views by using the proView Client, passes the received views to the Process Model View Wrapper part, creates the required internal data structures including mappings and builds the user interface elements. The building of the user interface elements is realized with the help of the vaadin MVC web framework. Therefore, the *VaadinRepresentation* implementation class extends the *VaadinApplication* class and uses the respective vaadin widget elements to generate a complex user interface. These vaadin dependent user interface implementation classes are summarized in the *Vaadin Componets* (**4.**) part. The user interface consists of two complex composite parts. One is a *NavigationMenu* that is used to indicate which of the respective process model parts are displayed in a content area. This content area contains a single *ComplexForm* UI element. An update of the content area and, thus, the form elements to display can be triggered by a selection in the NavigationMenu. This results in a call to the *refreshView* method in the Representation and finally in a redraw of the complex form area contents.

8.3 Results

The operation of the complex UI generation prototype can be summarized as described in the following steps:

1. Selection of a process model from a list of predefined, available ADEPT2 process models
2. Selection of a grouping criteria-specific process model as process view
3. These two selections trigger the complex UI generation which results in a complex user interface separated into two parts:
   a) A navigation menu, which is generated based on the control-flow block structure of the process model (process view)
   b) An area that contains user interface form elements in which the displayed form elements depend on the selection made by a user in the navigation menu

Figure 8.4 shows the result for a generated complex user interfaces that is based on *Use Case 3: Bank Account Creation*. The user interface and its single elements are based on widgets of the previously mentioned vaadin framework which, in turn, is based on Google Web Toolkit (GWT)[2] and Java web technology. In consequence, the UI generation prototype runs on a web server and requires a web browser to be accessed. In the following, a more detailed description of the numbered and highlighted regions of Figure 8.4 is given.

The region labeled with **1** and **2** of Figure 8.4 contains the selection mechanism to trigger a complex user interface generation. In this **1** is used to select a process model. These

[2]`http://code.google.com/webtoolkit/`, last checked January 30, 2012

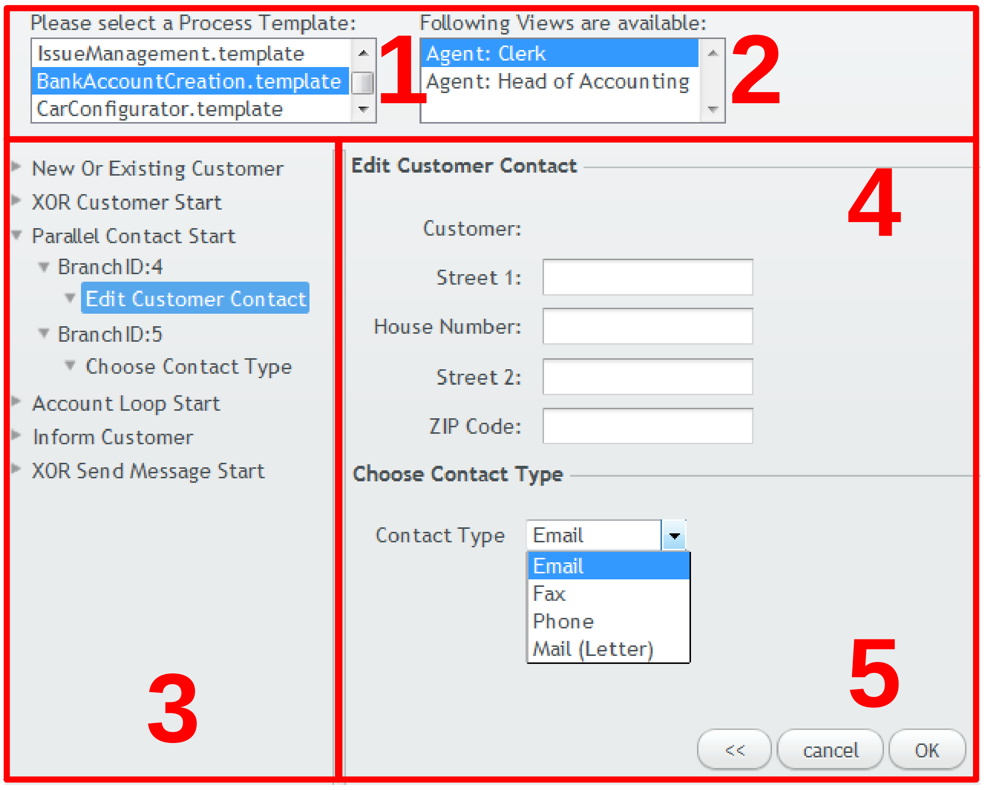
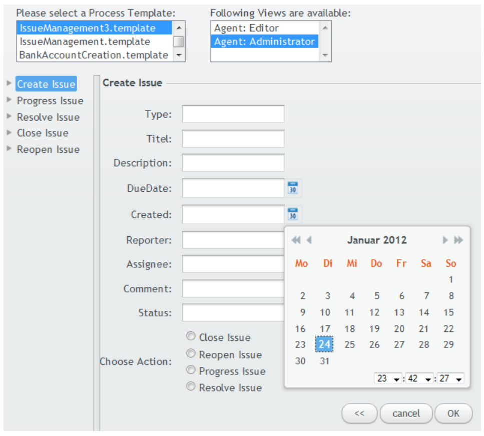
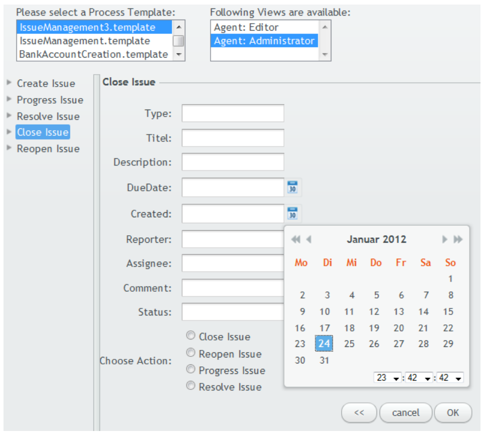


Figure 8.4: Regions in the Complex User Interface

process models are managed by the underlying proView and AristaFlow infrastructure of the prototype. After selecting one of the process models from the list in **1** the available process views (grouping criteria-specific process models grouped based on the used agents in the process model) for this process model are displayed in **2**. The selection of one of these views triggers the complex user interface generation mechanism. The results for this are displayed in the regions **3** and **4**. In which **3** is a interactive navigation menu based on the process models control-flow block structure and **4** the content area for generated complex user interfaces.

The navigation menu (**3**) is created based on the generated UI Model it is a direct representation of its tree-like data structure. It is possible to select elements in the navigation menu. If a element is selected the respective child elements are displayed in the content area **4**. The decision which elements to display is realized by a Mapping Meta Model implementation (cf. Chapter 6.2.1). With the help of this it is possible to detect the respective required activities in the process model (process view) and generate form elements based on their

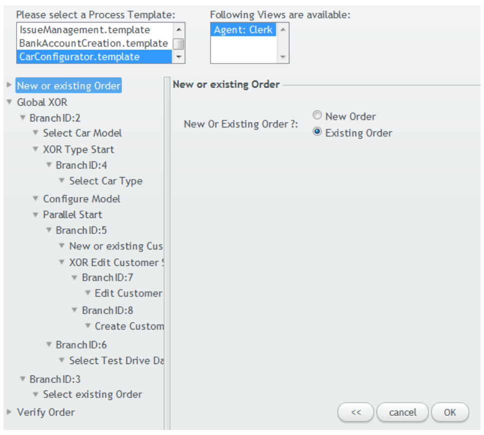


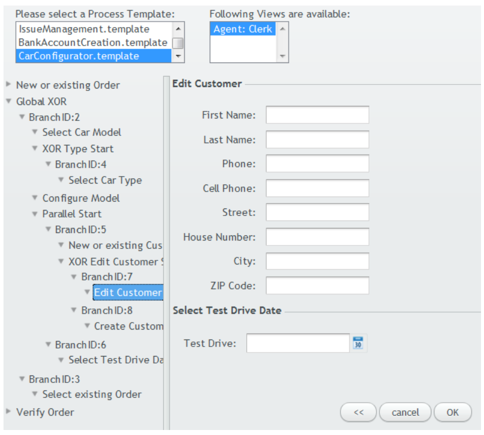


(a) Complex User Interface, State *Create Issue* (b) Complex User Interface, State *Close Issue*

Figure 8.5: Generated Complex User Interface Result, Use Case 1: Issue Management

data-flow. Region (**5**) (contained in the lower right of region (4)) represents the control-flow navigation element for process execution. After this basic introduction of the prototype in the following, the results for the generated UIs of the use cases are presented.



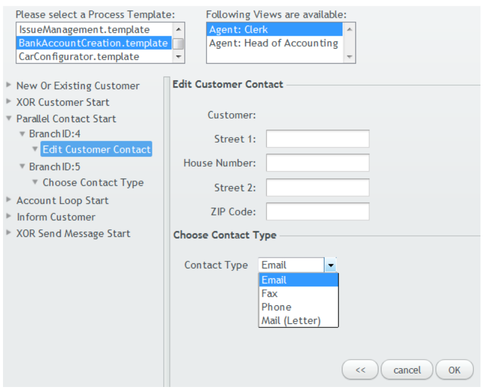


(a) Complex User Interface, State *New Or Existing Order* (b) Complex User Interface, State *Edit Customer*

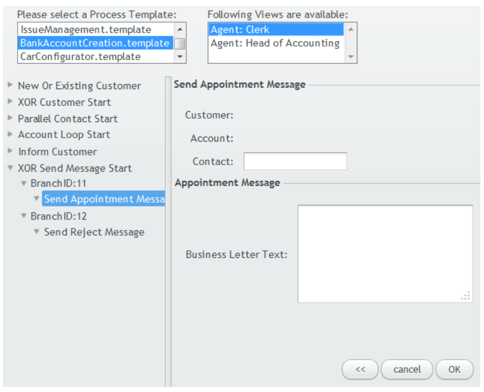
Figure 8.6: Generated Complex User Interface Result, Use Case 2: Car Configurator

Figure 8.5 shows two states for the generated complex UI-based on *Use Case 1: Issue Management*. In it, the process view for the agent *Administrator* has been selected. Figure 8.5a represents the user interface for the *Create Issue* activity and Figure 8.5b for the *Close Issue* activity. The only difference between those UIs is that the Create Issue UI contains input fields whereas the Close Issue UI contains edit fields. Since the UI generation mechanism is based on a process model there is no runtime data to display in the form fields and thus no visible difference between both UIs. In both UI states the field *Created* has been selected. Since the corresponding data element in the process model is a date element this results in displaying a supporting UI Widget to defined a date value.

Figure 8.6 shows two states for the generated complex user interface for *Use Case 2: Car Configurator*. Figure 8.6a shows the UI state for the *New or existing Order* activity whereas Figure 8.6 shows the UI state for *Edit Customer* activity. The Edit Customer activity is contained in a XOR block, this XOR block is, in turn, contained in a parallel block. Therefore the UI elements for the *Select Test Dirve Date* activity, which is contained in the different branch of the parallel block (referring to the New or existing Order activity), are also displayed. The complexity of the navigation menu element is a result of the nesting deep, of the underlying process model control-flow block structure. Since the process model contains only a single agent there is only one view available.



(a) Complex User Interface, State *Edit Customer Contact*



(b) Complex User Interface, State *Send Appointment Message*

Figure 8.7: Generated Complex User Interface Result, Use Case 3: Bank Account Creation

Figure 8.7 shows two states for the generated complex user interface for *Use Case 3: Bank Account Creation*. In this use case there exist two process views since the process model contains different agents. The sample states are based on the view for the agent *Clerk*. Figure 8.7a shows the UI elements for the *Edit Customer Contact* activity it is contained in a parallel block and thus the UI elements for the activity *Choose Contact Type* are also displayed. The *Choose Contact Type* UI elements are an example for the special handling of a list based data type (cf. Chapter 5.3.2). Based on a list of strings with predefined values a select UI element is generated. Figure 8.7b shows the UI elements for the activity

Send Appointment Message. Since this activity is contained in a selected branch of a XOR block only the UI elements for this activity are display. The division of the screen and the additional second heading is based on the fact that *Appointment Message* is a complex data element (cf. Chapter 5.9) for which an additional UI element group has been generated.

Based on these resulting generated complex user interfaces, in the next chapter the overall results of the thesis are summarized and discussed. In addition, further research questions are outlined, including potential approaches for the further development of the presented prototype.

9 Summary

The primary concern of the thesis was to show up the general feasibility of complex user interface generation based on process models. This concern was implemented and described, including various detailed aspects like advanced activity-grouping concerns. Nevertheless, there are still some remaining open questions in particular according to integration and run-time aspects of complex user interfaces. Therefore this concluding chapter presents a summary of the overall results in Chapter 9.1. This is followed by Chapter 9.2 which discusses further research questions that arise from the results and closes with an overall concluding statement in the final Chapter 9.3.

9.1 Results

The overall Transformation Model as presented in Chapter 5.1 builds a framework for the connection of the single distinct parts, which are required to implement an overall complex user interface generation based on process models. The first of this Transformation Model components to mention here are the *Elementary Transformation Patterns* which represent a systematic approach to describe the state of the art user interface generation for single process model activities based on the data-flow.

The second Transformation Model component is build by the *Complex Transformation Patterns*. These CTPs enforce one further step for complex user interface generation. An interpretation of process model control-flow structure and a subsequent mapping of this structure to user interface elements achieve this. In addition, these CTPs include some prospective patterns, which deal with usability aspects. In which they interpret certain process model fragments and transform them to complex UI elements that follow a predefined overall behaviour.

The last component of the Transformation Model is build by a Grouping Mechanism. It is based on process model metadata with special respect to organizational concerns. This enables the possibility to generate role-specific user interfaces. The Transformation Algorithm described in Chapter 6.2.3 uses the single components of the Transformation Model to realize a modular overall process model to complex user interface generation. With the help of the User Interface Model (cf. Chapter 3.2.2) and the Representation component (cf. Chapter 6.2.4) a flexible abstraction mechanism to decouple the overall transformation from technical related concerns, e.g., UI implementation specific aspects, is realized. Based on this, in Chapter 6.2.4 a detailed description for the delegation of changes from within a user interface to a related process model and vice versa has been described.

Complex user interface related run-time aspects including challenging problems have been discussed in Chapter 7. Based on the previously mentioned UI Model and the Representation component, the principle practicability of such user interface based (process instance) run-time changes has been outlined. The suggested solution uses the concepts of declarative process modelling, to ensure a correct execution of a process instance if a user interface based change has occurred. This aspect is supplemented with the introduction of a mechanism of how to deal with a retrospective modification of process instances control-flow from within a user interface. This retrospective modification uses the processed data elements to determine if a return from the currently activated activity to its predecessors is possible.

Referring to the complex user interface requirements defined based on use cases and research (cf. Chapter 4), it can be stated, up to a certain extend they were all addressed. Most of them, expect the UI-based modification options and retrospective control-flow modifications where prototypically implemented. Moreover, this prototypical implementation which has been described in Chapter 8 showed the general practicability of the overall process model to complex user interface transformation approach.

9.2 Further Research Questions

In this chapter, further arising research questions according to complex user interface generation are discussed. They are listed in the order of their appearance during the thesis.

With the Elementary Transformation Pattern *ETP2: Non-Human Activity Transformation* the basic requirement for handling service (non-human) activities according to an overall process model to user interface transformation have been described. For this ETP2 more specific details are necessary to realize a useable integration of the described UI generation approach in an existing BPMS. Without reasonable source code generation, as described by ETP2, all run-time related aspects cannot be implemented.

For the Complex Transformation Patterns a more detailed evaluation is necessary. This could be based on analyzing existing process models according to their user interface requirements, e.g., by examine the existing user interfaces for this models [WRR07, vdAtHKB03, LRtHW+11, LRWM+11, RvdAtHE05]. In addition, the existing process models can be rebuilt to fit in the described UI generation approach and then, in turn, the resulting generated complex user interface can be compared with the existing ones. The behaviour pattern aspect introduced as an extension to the structural CTPs is open field. An detailed analysis of existing process models should help to identify more patterns in this area (cf. Chapter 5.4.2).

The run-time aspects discussed in Chapter 7, like modification during run-time and sequence modification, should be included in the existing prototype. This would open up further research questions especially according to end user concerns. Before they are discussed in the following some general remarks according to the further development of the prototype. The prototype should not only generate complex user interfaces during process modelling, additionally it should serve as an advanced user interface template editor. In which user interfaces can be modified in a drag and drop manner. This can be enabled by the described change propagation mechanism (cf. Chapter 6.2.4). Based on this it even would be possible to create a process model, by creating the respective user interface.

Another important prototype related aspect is the connection of the generated complex user interfaces to a process execution engine. These aspects where somehow described by the previously mentioned ETP2. However, according to process execution more specific details are required. A starting point for this could be the definition of an *Interaction Service* for each generated UI, which encapsulates the non human activities in a service oriented manner by providing methods with respective input and output data as required by the human activities.

As mentioned before, a prototype enriched with the described model and run-time related aspects, would open up for various psychological and usability related analysis. For instance, evaluations according to time saving concerns in user interface development [SSR+07, ZZHM07], process execution speed up by generated complex UIs compared to standard UIs, practicability of complex UI generation in a real world scenario similar as described in

[SMV10] and last but not least tests according to the acceptance of run-time related complex UI features like UI-based changes during process execution.

9.3 Conclusion

In summary, it can be stated that it is possible to generate role specific complex user interfaces for complete block-structured process models. The introduced Transformation Model and pattern-based approach has proven to be feasible. Finally yet importantly the implementation of the overall Transformation Algorithm in the prototype represents a proof of concept for this user interface generation method.

This is somehow in contrast to statements from previous research work which outline the necessity of task models for process model based user interface generation [Sou09, GVC08]. However, some of these task model related aspects, like the granularity of activities in a process model, have been implicitly integrated in our approach (cf. Chapter 6.1). To bring this statements about complex user interface generation for process models to a further level. More efforts, especially according to the developed prototype and subsequent usability studies with real users are essential. Altogether the neglect aspect of complex user interface generation based on process models seem to be promising and worthwhile research direction. Thus the continuing of its development, e.g. along with the *proView* research project [RKBB12], is a desirable option.

Bibliography

[AD67] J. Annett and K. D. Duncan. Task Analysis And Training Design. Technical Report 1, Hull University (England) Departement of Psychology, 1967.

[Ben96] D. Benyon. Domain Models for User Interface Design. In D. Benyon and P. Palanque, editors, *Critical Issues in User Interface Systems Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[CNM83] S. K. Card, A. Newell, and T. P. Moran. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1983.

[CT12] M. Chinosi and A. Trombetta. BPMN: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134, 2012.

[DBVC09] R. Dividino, V. Bicer, K. Voigt, and J. Cardoso. Integrating Business Process and User Interface Models using a Model-driven Approach. In *Computer and Information Sciences, 2009. ISCIS 2009. 24th International Symposium on*, pages 492–497, September 2009.

[DR09] P. Dadam and M. Reichert. The ADEPT Project: A Decade of Research and Development for Robust and Flexible Process Support - Challenges and Achievements. *Computer Science - Research and Development*, 23:81–97, 2009.

[DRRM$^+$10] P. Dadam, M. Reichert, S. Rinderle-Ma, A. Lanz, R. Pryss, M. Predeschly, J. Kolb, L. T. Ly, M. Jurisch, U. Kreher, and K. Göser. From ADEPT to AristaFlow BPM Suite: A Research Vision Has Become Reality. In S. Rinderle-Ma, S. Sadiq, F. Leymann, W. M. P. van der Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, editors, *Proceedings Business Process Management (BPM'09) Workshops, 1st Int'l. Workshop on Empirical Research in Business Process Management (ER-BPM '09)*, volume 43 of *Lecture Notes in Business Information Processing*, pages 529–531. Springer Berlin Heidelberg, 2010.

[Dum05] M. Dumas, editor. *Process-Aware Information Systems : Bridging People and Software through Process Technology.* Wiley, Hoboken, NJ, 2005.

[EKO07] M. Ehrig, A. Koschmider, and A. Oberweis. Measuring Similarity between Semantic Business Process Models. In *Proceedings of the fourth Asia-Pacific conference on Comceptual modelling - Volume 67*, APCCM '07, pages 71–80, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[GBP$^+$01] T. Griffiths, P. J Barclay, N. W. Paton, J. McKirdy, J. Kennedy, P. D. Gray, R. Cooper, C. A. Goble, and P. P. da Silva. Teallach: A Model-Based User Interface Development Environment for Object Databases. *Interacting with Computers*, 14(1):31–68, 2001.

[GVC08] J. G. Garcia, J. Vanderdonckt, and J. M. G. Calleros. FlowiXML: a step towards designing workflow management systems. *Int. J. Web Eng. Technol.*, 4:163–182, May 2008.

[GVGW08] J. Guerrero, J. Vanderdonckt, J. M. Gonzalez, and M. Winckler. Modeling User Interfaces to Workflow Information Systems. In *Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems*, pages 55–60, Washington, DC, USA, 2008. IEEE Computer Society.

[Ham10] M. Hammer. What is Business Process Management? In J. vom Brocke and M. Rosemann, editors, *Handbook on Business Process Management 1*, International Handbooks on Information Systems, pages 3–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[HBR08] A. Hallerbach, T. Bauer, and M. Reichert. Managing Process Variants in the Process Life Cycle. In J. Cordeiro and J. Filipe, editors, *ICEIS (3-2)*, pages 154–161, 2008.

[HMZ11] H. Hussmann, G. Meixner, and D. Zuehlke. *Model-Driven Development of Advanced User Interfaces*, volume 340 of *Model-Driven Development of Advanced User Interfaces:, Studies in Computational Intelligence*, chapter Preface, pages V–VII. Springer-Verlag Berlin / Heidelberg, 2011.

[HRL09] O. Holschke, J. Rake, and O. Levina. Granularity as a Cognitive Factor in the Effectiveness of Business Process Model Reuse. In U. Dayal, J. Eder, J. Koehler, and H. Reijers, editors, *Business Process Management*, volume

5701 of *Lecture Notes in Computer Science*, pages 245–260. Springer Berlin / Heidelberg, 2009.

[HYZL10] A. Hongxin, X. Yusheng, M. Zhixin, and L. Li. Integrating User Interfaces by Business Object States. In *Information Science and Engineering (ICISE), 2010 2nd International Conference on*, pages 2900–2903, December 2010.

[Kam11] K. Kammerer. Technische Realisierung des Abklärungsablaufs von Sicherheitsausweisen in KMUs mit Hilfe der AristaFlow BPM Suite. Ulm University, June 2011.

[KLL09] R.K.L. Ko, S.S.G. Lee, and E.W. Lee. Business process management (BPM) standards: A survey. *Business Process Management Journal*, 15(5):744–791, 2009.

[KR11a] V. Künzle and M. Reichert. A Modeling Paradigm for Integrating Processes and Data at the Micro Level. In T. Halpin, S. Nurcan, J. Krogstie, P. Soffer, E. Proper, R. Schmidt, I. Bider, W. M. P. van der Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, and C. Szyperski, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 81 of *Lecture Notes in Business Information Processing*, pages 201–215. Springer Berlin Heidelberg, 2011.

[KR11b] V. Künzle and M. Reichert. PHILharmonicFlows: Towards a Framework for Object-aware Process Management. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(4):205–244, 2011.

[KRW12] J. Kolb, M. Reichert, and B. Weber. Using Concurrent Task Trees for Stakeholder-centered Modeling and Visualization of Business Processes. In *S-BPM ONE 2012*. Springer, April 2012.

[LK01] A. M. Latva-Koivisto. Finding a complexity measure for business process models. Technical report, Helsinki University of Technology, Systems Analysis Laboratory, 2001. 26 pp.

[LRD11] A. Lanz, M. Reichert, and P. Dadam. Robust and Flexible Error Handling in the AristaFlow BPM Suite. In P. Soffer, E. Proper, W. M. P. van der Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, editors, *Information Systems Evolution*, volume 72 of *Lecture Notes in Business Information Processing*, pages 174–189. Springer Berlin / Heidelberg, 2011.

[LRtHW$^+$11] M. La Rosa, A. H. M. ter Hofstede, P. Wohed, H. A. Reijers, J. Mendling, and W. M. P. van der Aalst. Managing Process Model Complexity via Concrete Syntax Modifications. *Industrial Informatics, IEEE Transactions on*, 7(2): 255–265, May 2011.

[LRWM$^+$11] M. La Rosa, P. Wohed, J. Mendling, A. H. M. ter Hofstede, H. A. Reijers, and W. M. P. van der Aalst. Managing Process Model Complexity Via Abstract Syntax Modifications. *Industrial Informatics, IEEE Transactions on*, 7(4): 614–629, November 2011.

[LV04] Q. Limbourg and J. Vanderdonckt. Comparing Task Models for User Interface Design. *The handbook of task analysis for human-computer interaction*, 6:135–154, 2004.

[LVM$^+$05] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero. USIXML: A Language Supporting Multi-path Development of User Interfaces. In R. Bastide, P. Palanque, and J. Roth, editors, *Engineering Human Computer Interaction and Interactive Systems*, volume 3425 of *Lecture Notes in Computer Science*, pages 134–135. Springer Berlin / Heidelberg, 2005.

[LW07] X. Lu and J. Wan. Model Driven Development of Complex User Interface. In A. Pleuß, J. V. den Bergh, H. Hußmann, S. Sauer, and D. Görlich, editors, *Model Driven Development of Advanced User Interfaces (MDDAUI 2007)*, volume 297 of *CEUR Workshop Proceedings*, pages 59–64. CEUR-WS.org, 2007.

[MPS02] G. Mori, F. Paternò, and C. Santoro. CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. *IEEE Trans. Software Eng.*, 28(8):797–813, 2002.

[MRvdA10] J. Mendling, H. A. Reijers, and W. M. P. van der Aalst. Seven process modeling guidelines (7PMG). *Information and Software Technology*, 52(2): 127–136, 2010.

[MS08] J. Mendling and M. Strembeck. Influence Factors of Understanding Business Process Models. In W. Abramowicz, D. Fensel, W. M. P. van der Aalst, J. Mylopoulos, M. Rosemann, M. J. Shaw, and C. Szyperski, editors, *Busi-*

ness Information Systems, volume 7 of *Lecture Notes in Business Information Processing*, pages 142–153. Springer Berlin Heidelberg, 2008.

[MSBS03] D. Moody, G. Sindre, T. Brasethvik, and A. Sølvberg. Evaluating the Quality of Process Models: Empirical Testing of a Quality Framework. In S. Spaccapietra, S. March, and Y. Kambayashi, editors, *Conceptual Modeling — ER 2002*, volume 2503 of *Lecture Notes in Computer Science*, pages 380–396. Springer Berlin / Heidelberg, 2003.

[OAS07] OASIS. Web Services Business Process Execution Language for Web Services version 2.0. OASIS: `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf`, 2007. Organization for the Advancement of Structured Information Standards.

[OMG11a] Object Management Group OMG. Business Process Model and Notation (BPMN). OMG: `http://www.omg.org/spec/BPMN/2.0/PDF/`, January 2011.

[OMG11b] Object Management Group OMG. Unified Modeling Language (UML). OMG: `http://www.omg.org/spec/UML/2.4.1`, August 2011.

[Pat00] F. Paternò. *Model-Based Design and Evaluation of Interactive Applications.* Springer London, 236 Gray's Inn Road, London WC1X 8HB United Kingdom, 1st edition. edition, 2000. 192 pp.

[PEGM94] A. R. Puerta, H. Eriksson, J. H. Gennari, and M. A. Musen. Model-Based Automated Generation of User Interfaces. In B. Hayes-Roth and R. E. Korf, editors, *AAAI*, pages 471–477. AAAI Press / The MIT Press, 1994.

[PM97] A. R. Puerta and D. Maulsby. MOBI-D: A Model-Based Development Environment for User-Centered Design. In *CHI '97 extended abstracts on Human factors in computing systems: looking to the future*, CHI EA '97, pages 4–5, New York, NY, USA, 1997. ACM.

[PMM97] F. Paternò, C. Mancini, and S. Meniconi. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In S. Howard, J. Hammond, and G. Lindgaard, editors, *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, volume 96 of *INTERACT '97*, pages 362–369, London, UK, UK, 1997. Chapman & Hall, Ltd.

[PS03] F. Paternò and C. Santoro. A Unified Method For Designing Interactive Systems Adaptable To Mobile And Stationary Platforms. *Interacting with Computers*, 15(3):349–366, 2003. Computer-Aided Design of User Interface.

[PvdA06] M. Pesic and W. M. P. van der Aalst. A Declarative Approach for Flexible Business Processes Management. In J. Eder and S. Dustdar, editors, *Business Process Management Workshops*, volume 4103 of *Lecture Notes in Computer Science*, pages 169–180. Springer Berlin / Heidelberg, 2006.

[RDB03] M. Reichert, P. Dadam, and T. Bauer. Dealing with Forward and Backward Jumps in Workflow Management Systems. *Software and Systems Modeling*, 2(1):37–58, 2003.

[Rei00] M. Reichert. *Dynamische Ablaufänderungen in Workflow-Management-Systemen*. PhD thesis, Ulm University, July 2000.

[RKBB12] M. Reichert, J. Kolb, R. Bobrik, and T. Bauer. Enabling Personalized Visualization of Large Business Processes through Parameterizable Views. In *27th ACM Symposium On Applied Computing (SAC'12), 9th Enterprise Engineering Track*. ACM Press, March 2012.

[RM08] H. Reijers and J. Mendling. Modularity in Process Models: Review and Effects. In M. Dumas, M. Reichert, and M. Shan, editors, *Business Process Management*, volume 5240 of *Lecture Notes in Computer Science*, pages 20–35. Springer Berlin / Heidelberg, 2008.

[RtHEvdA05] N. Russell, A. H. M. ter Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In L. Delcambre, C. Kop, H. Mayr, J. Mylopoulos, and O. Pastor, editors, *Conceptual Modeling – ER 2005*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368. Springer Berlin / Heidelberg, 2005.

[RvdAtHE05] N. Russell, W. M. P. van der Aalst, A. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In O. Pastor and J. Falcão e Cunha, editors, *Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 11–42. Springer Berlin / Heidelberg, 2005.

[SMV10] K. S. Sousa, H. Mendonça, and J. Vanderdonckt. A Rule-Based Approach for Model Management in a User Interface – Business Alignment Framework. In

D. England, P. Palanque, J. Vanderdonckt, and P. Wild, editors, *Task Models and Diagrams for User Interface Design*, volume 5963 of *Lecture Notes in Computer Science*, pages 1–14. Springer Berlin / Heidelberg, 2010.

[Sou09] K. S. Sousa. Model-Driven Approach for User Interface: Business Alignment. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '09, pages 325–328, New York, NY, USA, 2009. ACM.

[SSR$^+$07] N. Sukaviriya, V. Sinha, T. Ramachandra, S. Mani, and M. Stolze. User-Centered Design and Business Process Modeling: Cross Road in Rapid Prototyping Tools. In C. Baranauskas, P. Palanque, J. Abascal, and S. Barbosa, editors, *Human-Computer Interaction – INTERACT 2007*, volume 4662 of *Lecture Notes in Computer Science*, pages 165–178. Springer Berlin / Heidelberg, 2007.

[SV11] K. S. Sousa and J. Vanderdonckt. Business Performer-Centered Design of User Interfaces. In Hussmann et al. [HMZ11], chapter Preface, pages 123–142.

[Szw11] G. Szwillus. Task Models in the Context of User Interface Development. In H. Hussmann, G. Meixner, and D. Zuehlke, editors, *Model-Driven Development of Advanced User Interfaces*, volume 340 of *Studies in Computational Intelligence*, pages 277–302. Springer-Verlag Berlin / Heidelberg, 2011.

[TKVW10] V. Tran, M. Kolp, J. Vanderdonckt, and Y. Wautelet. Using Task and Data Models for user Interface Declarative Generation. In J. Filipe and J. Cordeiro, editors, *ICEIS (5)*, pages 155–160. SciTePress, 2010.

[TMN04] H. Trætteberg, P. J. Molina, and N. J. Nunes. Making Model-Based UI Design Practical: Usable and Open Methods and Tools. In *Proceedings of the 9th international conference on Intelligent user interfaces*, IUI '04, pages 376–377, New York, NY, USA, 2004. ACM.

[vdAtHKB03] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14:5–51, 2003.

[vdAtHW03a] W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske. Business Process Management: A Survey. In Weske [vdAtHW03b], pages 1019–1019.

[vdAtHW03b] W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske, editors. *Business Process Management, International Conference, BPM 2003, Eindhoven, The Netherlands, June 26-27, 2003, Proceedings*, volume 2678 of *Lecture Notes in Computer Science*. Springer, 2003.

[VVK08] J. Vanhatalo, H. Völzer, and J. Koehler. The Refined Process Structure Tree. In M. Dumas, M. Reichert, and M. Shan, editors, *Business Process Management*, volume 5240 of *Lecture Notes in Computer Science*, pages 100–115. Springer Berlin / Heidelberg, 2008.

[W3C07] World Wide Web Consortium W3C. XSL Transformations (XSLT) Version 2.0. W3C: `http://www.w3.org/TR/xslt20/`, January 2007.

[Wes07] M. Weske. *Business Process Management : Concepts, Languages, Architectures*. Springer, Berlin; Heidelberg; New York, September 2007. 368 pp.

[WRR07] B. Weber, S. Rinderle, and M. Reichert. Change Patterns and Change Support Features in Process-Aware Information Systems. In J. Krogstie, A. Opdahl, and G. Sindre, editors, *Advanced Information Systems Engineering*, volume 4495 of *Lecture Notes in Computer Science*, pages 574–588. Springer Berlin / Heidelberg, 2007.

[WSR09] B. Weber, S. Sadiq, and M. Reichert. Beyond Rigidity – Dynamic Process Lifecycle Support: : A Survey on Dynamic Changes in Process-aware Information Systems. *Computer Science - Research and Development*, 23:47–65, 2009.

[WWWC08] W3C World Wide Web Consortium. Extensible Markup Language (XML). W3C: `http://www.w3.org/TR/2008/REC-xml-20081126/`, November 2008.

[XJ07] Lu X. and Wan J. User interface design model. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, volume 3, pages 538–543, August 2007.

[YSW$^+$10] S. Yang, Y. Sun, J. Waterhouse, D. Lau, and T. Al-Hamwy. Modeling and Implementing a Business Process Using WebSphere Lombardi Edition 7.1. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '10, pages 374–375, Riverton, NJ, USA, 2010. IBM Corp.

[ZZHM07] X. Zhao, Y. Zou, J. Hawkins, and B. Madapusi. A Business-Process-Driven Approach for Generating E-Commerce User Interfaces. In G. Engels, B. Opdyke, D. Schmidt, and F. Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 256–270. Springer Berlin / Heidelberg, 2007.

A Appendix

A.1 User Interface Generation Approaches

		UI Bussines Alignment (Sousa et al.)			Philharmonic Flows (Künzle et al.)		ConcurTaskTrees (Paterno et al.)	
Conceptual Steps	1.	List of Mapping Busines Process to Task Model Elements			1.	derivate hierarchical process models from data object processing in information systems by :	1.	Decied How to Create Presentations for Tasks (1. Same for All Task Types, 2. Different for all Task Types, Combination of 1. & 2.)
		Basend on BPMN , Separated into Mappings for:				1. Macro Processes = Data Object Interaction	2.	Identifiy which Task can be grouped in one User Interface by checking which Tasks can be enabeld at the same time
		1. Bussines Elements to Task Elements						
		2. Activity Attributes to Taks Properties				2. Micro Processes = Data Object Behvaior	3.	Decied How to Arrange the Functionality of theses Tasks in a UI by considering Data and Control Flow Connections
		3. Process Activities and Task Types						
	2.	Four Development Steps:	Based on UsiXML / FlowiXML		2.	Prerequisite: Detailed Data Type Model	4.	Validate the Set of enabled Tasks about Contained Task Patterns
		1. Creat Conceptual Models (Task Models, Data Models, User Models)				For Each Data Type: define a Micro Process (= finite state machine)	5.	Choosing Presentation Template based on Task Semantic
		2. Create Abstract UI (AUI) using UsiXML and the Conceptual Models				Connection of DataInstances (Micro Proecesses) = Macro Process	6.	Considering Temporal Relation between Tasks (This Seems to be a good Input for the CTPs (cf. Paterno Model Based Design Book p 82.)
		3. Create Concret UI (CUI) based on AUI				Rolse specific Acces on Macro & Micro Level defined in a Authorisation Tabel	7.	Processing The connections between multiple Enabled Task Sets (= CTP Pattern nesting & Block Processing)
		4. Create Final UI (FUI) based on CUI (e.g. html, swing etc.)				Authorisation Tabel is used to generate Role Specific Forms, by using data type & data access based transformation rules	8.	Ordering UI Elements for Processed Data based on their priority (level in the cct)
Implementation	BP Model	BPMN			No information about the concret steps taken could be found. The basic principal is as described in the last conceptual step:		1.	Use Arichtectural Model to describe UI Components (Consists of Several Sub Models e.g. MVC, or model to describe interactions)
	Task Model	ConcurTaskTrees (Global Task Model Based on Complete BP Model, with Sub Task Models based on BP Activities)						
	Abstract UI	User Interface without Representation Form based on CTT Model	UsiXML (CAMELEON Reference Framework default Implementation), FlowiXML process/ Workflow Extension for UsiXML		The Authorisation Table is ued to gerate Role Specific Forms for the processing of data objects. To achieve this data type and data access based transformation rules are applied		2.	Transformation From Task Model to UI Architekture Model = Map Tasks to SW UI Objects (interactors) cf. Paterno Model Based Design Book p. 115 R1) -R3)
	Concret UI	AUI + Representation Form (default form is 'graphical'), Plattform indetent		OWLAPI (OWL Ontology Processing); SWRL Bridge (Transform SWRL Rules in OWL); Drools (Manage Models)			3.	Connect the Interactors to support Information Flow, cf. Paterno Model Based Design Book p, 116 R4) - R19)
	Final UI	Implementation of CUI by transformation to e.g. HTML					4.	Task Patterns describe Common UI Usecases e.g. "search", Theses UsesCases imply a certain Task Structure connected with a certain UI design

Table A.1: From Model to User Interface Elements, Approach Comparison

A.2 Sample Process Models

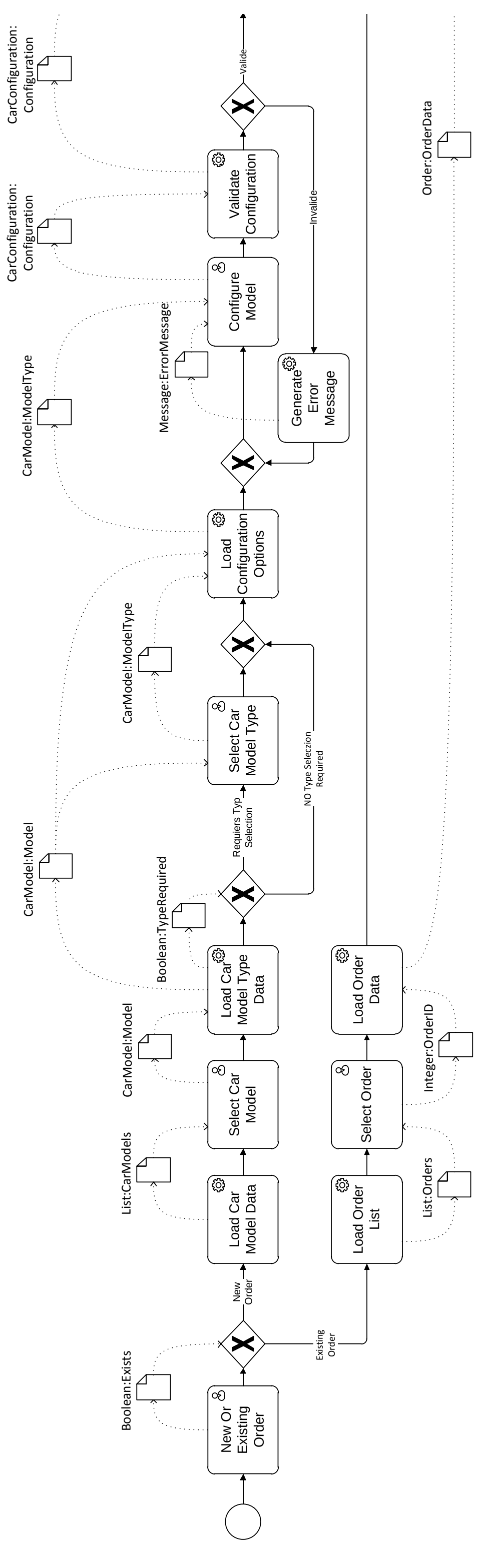


Figure A.1: Process Model Example: Details Car Configurator , first part

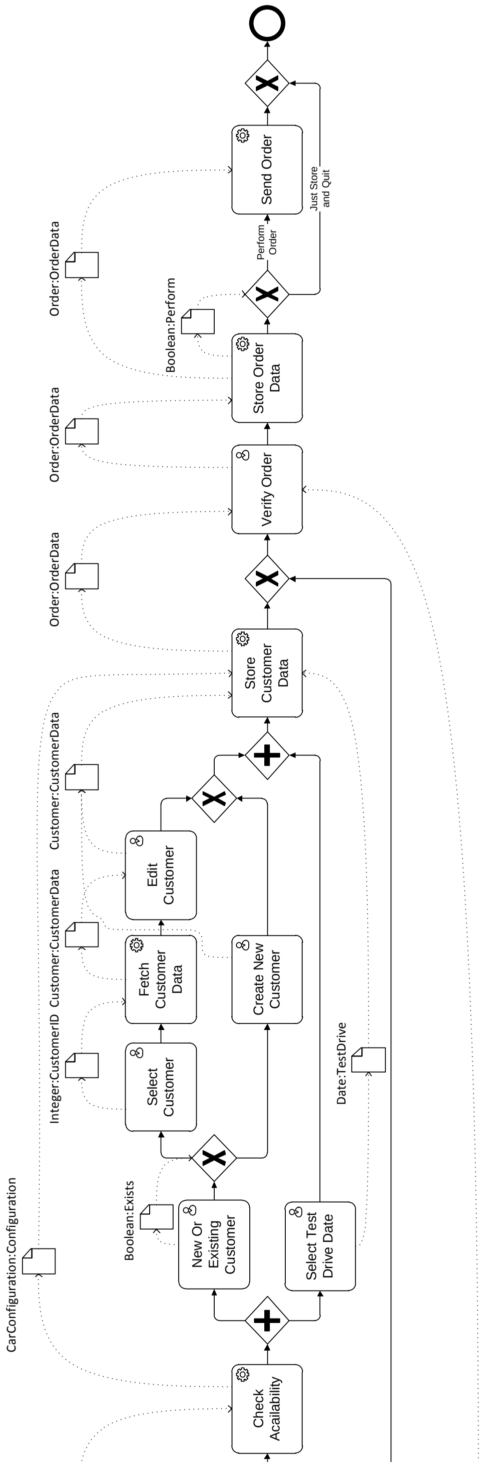


Figure A.2: Process Model Example: Details Car Configurator, second part

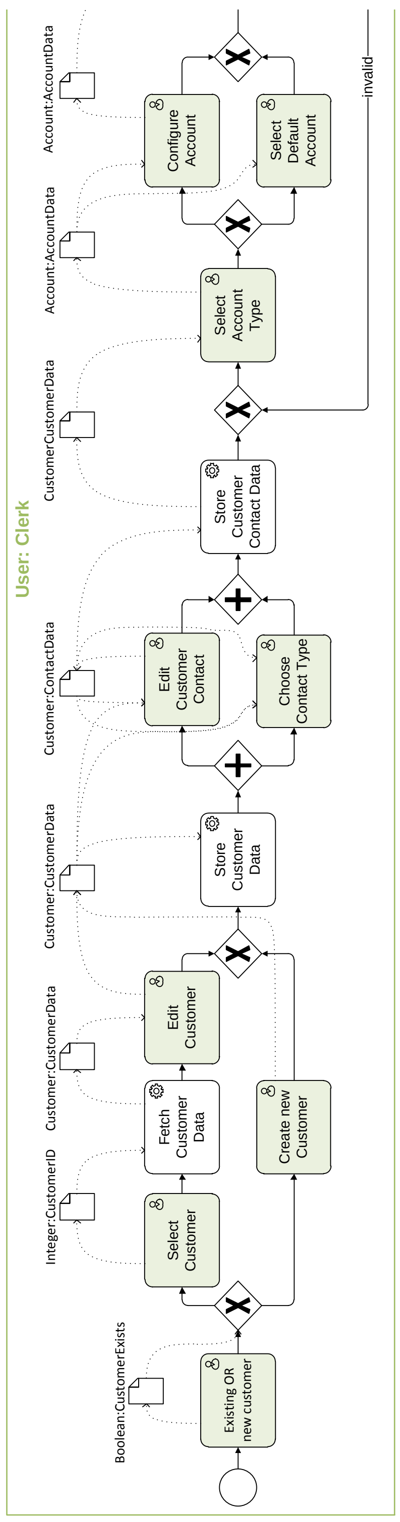


Figure A.3: Process Model Example: Details Bank Account Creation with different Roles highlighted, first part

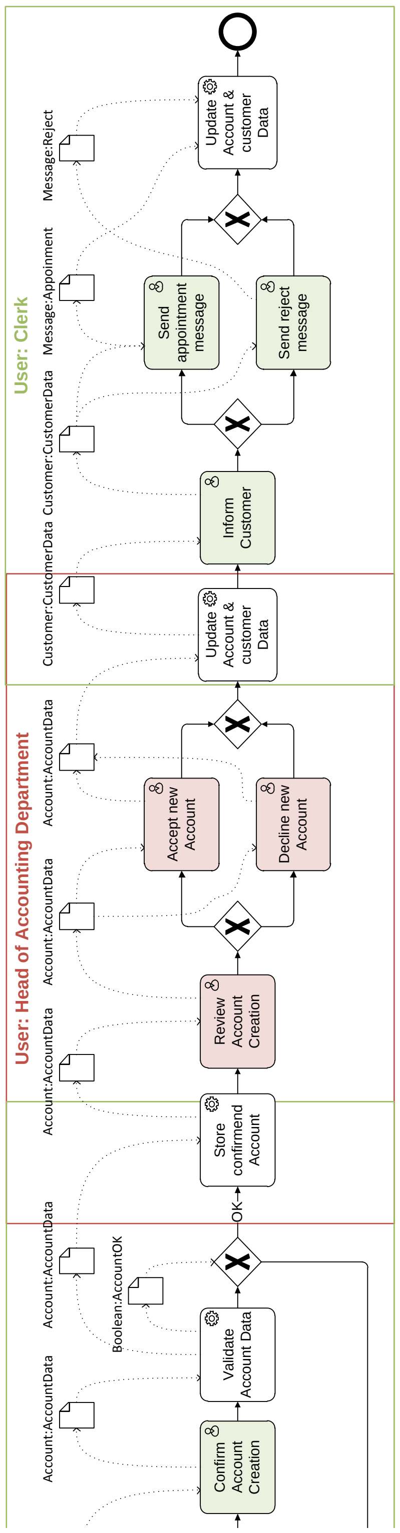


Figure A.4: Process Model Example: Details Bank Account Creation with different Roles highlighted, second part

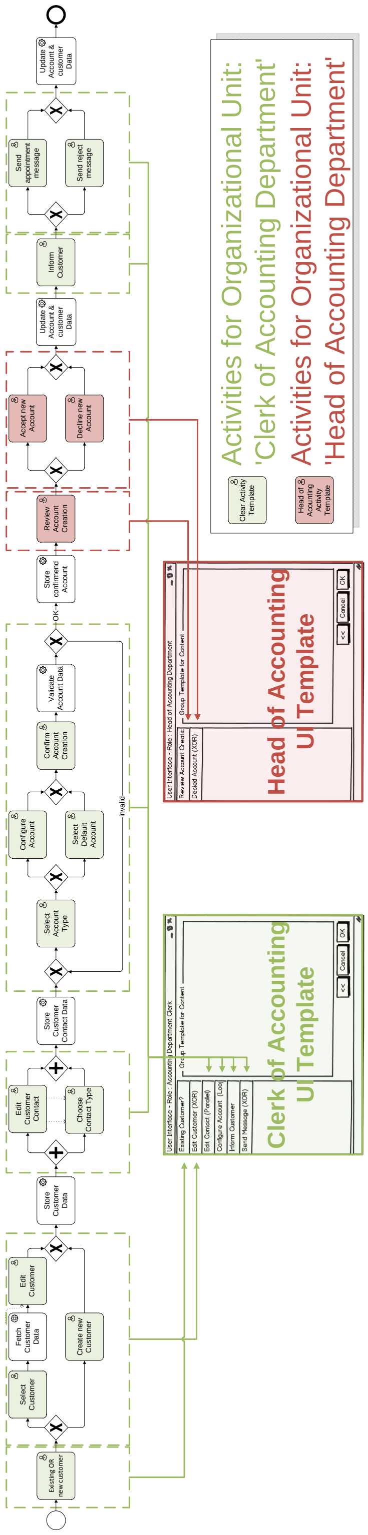


Figure A.5: Process Model Example: Bank Account Creation, Role-specific UI Allocation

Name: Paul Hübner Matrikelnummer: 708619

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ....

Paul Hübner