transnationale Universiteit Limburg
School voor Informatietechnologie

Dynamic User Interface Generation for Mobile and Embedded Systems with Model-Based User Interface Development

---

Proefschrift voorgelegd tot het behalen van de graad van

Doctor in de Wetenschappen, richting Informatica

aan de transnationale Universiteit Limburg

te verdedigen door

Kris Luyten

Promotor: Prof. dr. Karin Coninx

2000 – 2004

Acknowledgments

Research is never done alone. On the one hand a researcher is inspired by the work and findings of others, and on the other hand he is influenced by his colleagues. I have been in the fortunate position to work in a team of researchers sharing the same scientific interests as I have. The “Gebruikersgerichte Systeemontwikkeling” team in which I had the opportunity to work, was lead by prof. dr. *Karin Coninx* who also was my adviser for this work. I am grateful and honored to be the first PhD student under her guidance. None of this work would exist without her support and open minded approach. If this was an advertisement, I would advise every PhD student to solicit for her guidance.

The other members were *Chris Vandervelpen*, *Bert Creemers*, *Jan Van den Bergh*, *Jo Segers* and *Tim Clerckx*. The original team grew out of the SEESCOA[1] project. Most of them also contribute to the CoDAMoS[2] project now, some of them pursue other goals. A big thanks to all of them, because they are an important reason why this work got completed and provided with me with feedback on early drafts of this text.

Prof. dr. *Eddy Flerackers*, managing director of the EDM, prof. dr. *Wim Lamotte*, prof. dr. *Philippe Bekaert* and prof. dr. *Frank Van Reeth*, professors at the EDM, were also from invaluable support. Prof dr. *Frank Neven* from the research group on theoretical computer science provided us with very useful feedback and pointers for some of the more formal notations we use in this dissertation.

Other people not in the team were also very helpful, sometimes directly related to our research work, sometimes not so directly. I want to thank

[1] Software Engineering for Embedded Systems using a Component-Oriented Approach, FWO/IWT project 980374, `http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/SEESCOA/`

[2] Context-driven Adaptation of Mobile Services, SBO/IWT project 030320 `http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/CoDAMoS/`

especially *Tom Van Laerhoven*, *Peter Quax* and *Jori Liesenborgs* who provided me with lots of technical support. I am also grateful to my other colleagues *Koen Beets*, *Erwin Cuppens*, *Joan De Boeck*, *Tom De Weyer*, *Fabian Di Fiore*, *Jan Fransens*, *Erik Hubo*, *Tom Jehaes*, *Pieter Jorissen*, *Tom Mertens*, *Patrick Monsieurs*, *Chris Raymaekers*, *Daniel Teunkens*, *William Van Haevre* and *Peter Vandoren*. Some parts of this work have been influenced greatly by them, other parts were developed together with them. I will do my very best to give the appropriate credits to the people that helped me throughout the actual text.

I am also grateful to the EDM secretariat, *Ingrid Konings*, *Heidi De Winter* and *Roger Claes* for all the things they arranged for me during the past four years.

On a personal note, I am in great debt to my *parents*, Jos and Anita, and my *sister* Sofie: they supported my studies and gave me sufficient freedom to make my own decisions. Also thanks to all my friends for the necessary diversions I needed during my studies and research work. And finally, I could not manage to live life so easily without *Kristel Clijsters*, who has been on my side for seven years now. She supported me tremendously and sacrificed as much as I did due to my fixation on all things related to work. Thank you Kristel, for making my life easier than it should be!

Abstract

A user's physical and virtual environments are becoming increasingly interwoven. Mobile and embedded devices have become more the rule than the exception: many present-day users are inexperienced when it comes to traditional computers, but are in daily contact with computerized systems. The increasing diversity of all kinds of devices, together with the possibilities of running arbitrary software autonomously, result in a thousand-and-one potential areas of use, whether mobile or not. The possibility of communicating with the (in)direct environment using other devices and observing that same environment allow us to develop ambient intelligent software which has knowledge of the environment and of the usage pattern of this software. Despite the support for software development for this kind of applications, some gaps still exist, making the creation of consistent, usable user interfaces more difficult.

The design methodologies for interactive systems need to support this new situation: an interactive software system is no longer designed for one particular piece of hardware or a single context-of-use. We turn to Model-Based User Interface Development and investigate what needs to be done to support the design and creation of interactive systems that can be used in multiple contexts. Model-Based User Interface Development uses a selected set of models to describe different aspects of a user interface such as user, domain, task, dialog and presentation. We select three models that are widely accepted; the task model, the dialog modal and the presentation model, and introduce the necessary enhancements to serve our purpose.

We noticed the lack of flexibility and scalability of the current methodologies using one or more of our selected models. In our approach, Dygimes, we show a framework and methodology to have a concrete and clear user interface design and creation cycle which covers all aspects from task modeling to the concrete user interface. The different models are annotated, transformed, linked and mapped, so they can smoothly integrate with each other, resulting

in a user interface that is scalable for a wide range of devices.

We demonstrate a tool chain built upon the Dygimes framework supporting the annotation, transformation, linking and mapping of models. Unlike other approaches this tool chain maximizes the automatic consistency checks between the different models and the models and the user interfaces without introducing a new formal (declarative) language to do so. The aim is to provide simple building blocks that are easy to understand but are still powerful enough to produce a flexible and scalable user interface that is suitable for multiple contexts-of-use. For this reason we extended a language for task modeling, designed a simple XML-based language for the presentation model and implemented algorithms that assist the designer in creating consistent dialogs and navigation of the user interface according to the abstract models she/he specified.

The evolution towards ubiquitous systems is at present governed by a shift from desktop computers towards mobile and embedded devices. With the framework we present in this dissertation we believe we have opened the road towards an enhanced way of Model-Based User Interface Development that supports the new requirements to build interactive systems that are suitable for this kind of devices. The framework does not only take into account new characteristics of modern computing platforms, it is also suitable to make the transition towards desiginging the interactive part of ubiquitous systems by supporting context-awareness.

Contents

Acknowledgments **iii**

Abstract **v**

Contents **ix**

List of Figures **xiii**

List of Listings **xv**

List of Tables **xvii**

I User Interface Creation for Mobile and Embedded Systems **1**

1 Introduction **3**

1.1 Problem Statement 3
1.2 Motivation and Aims 4
1.3 Overview 6

2 Model-Based User Interface Development **9**

2.1 Introduction 9
2.2 Model-Based User Interface Development 10
2.2.1 A Definition and some History 10
2.2.2 A More Precise Definition of Models 16
2.2.3 Our Selected Models 18
2.3 The Task Model 19
2.4 The Dialog Model 23

2.5 The Presentation Model 25
2.6 Model Relations and Mappings 27
2.7 Plasticity and Context in Models 28
2.8 Discussion 29

3 High-Level User Interface Description Languages 31
3.1 Introduction 31
3.2 History of XML-based User Interface Description Languages . . 33
3.3 An overview of XML-based High-Level User Interface Description Languages 35
3.3.1 Abstract User Interface Markup Language (AUIML) . . 35
3.3.2 Renderer Independent Markup Language (RIML) 36
3.3.3 Useware Markup Language (useML) 37
3.3.4 Teresa XML 38
3.3.5 Interface Specification Meta-Language (ISML) 39
3.3.6 The User Interface Markup Language (UIML) 40
3.3.7 XIML 41
3.3.8 UsiXML 42
3.4 Discussion 43

II HCI Engineering, Models and Transformations 49

4 Dygimes: Dynamically Generating Interfaces for Mobile and Embedded Systems 51
4.1 Introduction 51
4.2 Dygimes process 53
4.3 XML-based User Interface Descriptions 54
4.4 Task Model 56
4.5 The System Glue: an Interaction and Application Model 60
4.6 Automatic Layout Management 63
4.7 Customization and Templating 63
4.8 Towards a Tool Chain to support Model-Based User Interface Development 65
4.9 Discussion 68

5 Models for Multi-Device User Interfaces 69
5.1 Introduction 69
5.2 Related Work 70
5.3 The Task Model within the Dygimes Framework 72

5.4 ConcurTaskTrees formalism 73
5.5 An algorithm to calculate enabled task sets 77
5.5.1 Introduction 77
5.5.2 Generating a priority tree 78
5.5.3 Calculating the enabled task sets 80
5.6 Activity Chain Extraction 84
5.7 Dynamic Behavior of the User Interface 85
5.7.1 Mapping Sets on States 87
5.7.2 Finding the Initial State 87
5.7.3 Detecting Transitions 87
5.7.4 Mapping the Finishing States 90
5.7.5 The resulting State Transition Network 91
5.8 Actual transitions between dialogs 93
5.9 Discussion 94

6 Presentation of the User Interface 95
6.1 Introduction 95
6.2 Towards an XML-based HLUID Language 97
6.3 A Declarative Language for User Interface Design 98
6.4 SEESCOA XML 100
6.5 UiBuilder: A SEESCOA XML Renderer 105
6.6 Event handling in SEESCOA XML 110
6.7 Discussion 113

7 Multi-device Layout Management 115
7.1 Introduction 115
7.2 Related Work 116
7.3 Constraint Satisfaction and Layout Management 118
7.4 Calculating Presentation Structures 120
7.4.1 Describing spatial constraints 120
7.4.2 Building the layout description graph 121
7.4.3 Calculating widget positions 121
7.4.4 Conflict handling 122
7.4.5 Further screen space reduction strategies 123
7.5 Discussion 123

8 Components and Multi-Device User Interfaces 127
8.1 Introduction 127
8.2 Component-Based Software Development 129

8.3 User Interface Descriptions and Components 130
8.3.1 The SEESCOA Component Framework 130
8.3.2 The Rendering Component 134
8.3.3 A Case Study: a Camera Surveillance system 135
8.3.4 Decomposing tasks: relating components to tasks 137
8.4 Discussion 140

9 Uiml.net: an Open Uiml Renderer for the .Net Framework 143
9.1 Introduction 143
9.2 UIML Overview 145
9.3 Related Work 148
9.4 The Renderer 149
9.4.1 Overall Design 149
9.4.2 Dynamic Core 151
9.5 Inter-vocabulary distances 153
9.6 The Layout Problem 157
9.7 UIML and Dygimes 159
9.7.1 Integration with the task specification 161
9.7.2 Generation of the dialog model 161
9.8 Discussion 164

III Towards Context-Sensitive Model-Based User Interface Development 167

10 Extending Dygimes for Context-Sensitive User Interface Development 169
10.1 Introduction 169
10.2 Related Work 170
10.3 Dygimes Once Again 171
10.4 Design Process 172
10.4.1 The Context-Sensitive Task Model 173
10.4.2 The Presentation Model 175
10.5 A Case Study: Manage Stock 177
10.6 Discussion 183

11 Future Work 185
11.1 Dynamic Model-Based User Interface Development 186
11.2 Distributed User Interfaces 187
11.3 Next-Generation Widget Toolkits 187

11.4 Software Engineering 188

12 Conclusions 189

12.1 Model-Based User Interface Development 189
12.2 Achievements and Main Contributions 190
12.3 Scientific Contributions and Publications 191
12.4 Concluding Remarks... 193

A Scenarios 195

A.1 Scenario 1: Teaching with Technology 195
A.2 Scenario 2: Mobile Communication 196
A.3 Scenario 3: A Mobile Tourist Guide in a Museum 198
A.4 Technological Challenges 201

B Nederlandstalige Samenvatting 203

B.1 Inleiding 203
B.2 Model-gebaseerde Gebruikersinterface Ontwikkeling 204
B.3 Dygimes: Dynamische Generatie van Interfaces voor Mobiele en Ingebedde Systemen 205
B.4 Modellen voor Interface Ontwerp voor meerdere Apparaten . . 207
B.5 Presentatie van de Gebruikersinterface 207
B.6 Layoutbeheer voor Meerdere Apparaten 209
B.7 Componenten en Gebruikersinterfaces voor Meerdere Apparaten 210
B.8 Uiml.net: een Open Uiml Renderer voor het .Net Raamwerk . 211
B.9 Context in de Ontwikkeling van Gebruikersinterfaces 212
B.10 Besluit 213

Bibliography 229

List of Figures

1.1 Hardware setup with Dygimes UiBuilder 6

2.1 Architecture of Model-Based User Interface Development environments 14
2.2 An example task specification for getting a soft drink. 20
2.3 An example dialog specification from [DFAB04] 24

3.1 Interactor coverage versus number of interactor specific tags . . 44

4.1 The Dygimes process 54
4.2 The login dialog, rendered for several devices 57
4.3 Managing a simple publication database 58
4.4 The ConcurTaskTrees annotation tool 59
4.5 The location-transparent action handling glue 62
4.6 Three possible mappings for the query result on the AWT platform 65
4.7 Managing spatial constraints 67

5.1 The Dygimes User Interface design and generation process . . . 74
5.2 Mobile phone interaction design 76
5.3 A ConcurTaskTrees tree and its priority representation 79
5.4 A simple email client task specification. 83
5.5 A simple email client task specification annotated with user interface building blocks. 83
5.6 ConcurTaskTrees concurrency in a STN 89
5.7 $Task_{Start\ ID\ Form}$ is a leaf and $Task_{Perform\ Query}$ is no leaf. . . 90
5.8 $Task_{Submit}$ is a leaf and $Task_{Select\ File}$ is no leaf. 90
5.9 Neither $Task_{Personal\ Info}$ or $Task_{Job\ Info}$ are leaves. 91
5.10 Extracting the STN when a disabling relation is involved . . . 92

5.11 The state transition network for the email application 92
5.12 The state transition network for the email application 93

6.1 AIO versus CIO 101
6.2 Camera-based surveillance system using UiBuilder 106
6.3 UiBuilder Core Class Hierarchy 109
6.4 Communicating range interactors rendered from a SEESCOA XML description 112
6.5 User Interface with Python support rendered in AWT from listing 6.6 113

7.1 A visual representation of the constraint definition 119
7.2 The calculation of the presentation structure 122
7.3 A multi-device hotel registration form 124

8.1 SEESCOA component design example 131
8.2 Surface, internal and rendering components 132
8.3 Component composition for a camera surveillance system . . . 135
8.4 The mosaic component 136
8.5 The Mosaic component on a desktop 137
8.6 The Mosaic component on a PDA 138
8.7 A CTT diagram: checking for burglars (context-sensitive) . . . 139
8.8 A CTT diagram: checking for burglars 140

9.1 The UIML Meta-Interface Model 146
9.2 The dictionary example in UIML 148
9.3 A rough sketch of the Uiml.net architecture 151
9.4 Processing an UIML file with Uiml.net 152
9.5 Differences between Gtk# and SWF interface shown by the Meld tool 156
9.6 Copy Text example with UIML 157
9.7 A calculator on multiple platforms and with multiple widget sets158
9.8 The Multi(ple)-device Picture Browser with UIML 160
9.9 Layout constraints for the controls of figure 9.8 161

10.1 Context-Sensitive user interface Design Process 173
10.2 Context-Sensitive Task Model of the *Manage Stock* example . . 177
10.3 *Overview PDA* subtree 178
10.4 *Update PC* subtree 179
10.5 Context-Specific Task Model 180

10.6 Dialog Model for the stock example 181
10.7 Dialog Model with the concrete dialogs 182

List of Listings

4.1 The login dialog user interface description 55
4.2 The binding between an abstract user interface and the application logic 61
4.3 Two of the specified mapping rules 64
6.1 The SEESCOA High-Level User Interface Description Language Schema. 101
6.2 An example SEESCOA XML listing for a camera. Developed for the SEESCOA researcht project (IWT 980374) in cooperation with other partners. 103
6.3 A date group 105
6.4 SEESCOA XML description for a motion detection software component. Developed for the SEESCOA researcht project (IWT 980374) in cooperation with other partners. 106
6.5 Communicating range interactors in SEESCOA XML 111
6.6 Python support in SEESCOA XML 111
7.1 A SEESCOA XML constraint description for a group of AIOs. 120
8.1 user interface description of a single camera component 133
8.2 user interface description of a Mosaic component 133
9.1 The UIML code for the Dictionary example depicted in 9.2. . . 146
10.1 Decision DTD 174
10.2 Decision XML example 174
10.3 Decision rules for the *Overview PDA* task 177
10.4 Decision XML for the *Use Properties* task 178

List of Tables

3.1 Overview of selected properties of XML-based High-Level User Interface Description Languages 46
3.2 A model-based comparison of XML-based High-Level User Interface Description Languages. 47

5.1 The priority order of ConcurTaskTrees temporal operators. . . 78

9.1 User Interface building blocks for each leaf task from figure 5.9(a) 162
9.2 Two examples of merging UIML building blocks from an enabled task set with a predefined container. 166

Part I

User Interface Creation for Mobile and Embedded Systems

Chapter 1

Introduction

Contents

1.1 Problem Statement **3**
1.2 Motivation and Aims **4**
1.3 Overview **6**

1.1 Problem Statement

With the increasing heterogeneity of computing environments, developing interactive systems for these environments requires more work. A precondition of a good problem analysis is to have a clear but compact problem statement. If we capture the problem we try to solve in one sentence this could be as follows:

The lack of support for the design, development and deployment of multi-device user interfaces with an emphasis on design and development techniques that can be easily reused separately and independently.

The best way to identify the aims of this dissertation is to introduce some concrete problem scenario's that pose challenges for user interface design with current techniques and methodologies. Inspired by the ISTAG scenarios[1]

[1] Information Society Technologies Advisory Group, `http://www.cordis.lu/ist/istag.htm#istag-ambientintelreport`

[DBS+01] we provide three scenarios in appendix A that show how the evolution in technology will challenge the design of interactive systems in the following years. The first scenario in section A.1, emphasizes how mobile devices and embedded systems can cooperate and make use of *migratable user interfaces*. Migratable user interfaces are interfaces that can travel from one device to another offering the same functionality while changing its own concrete presentation. The second scenario in section A.2, is developed in cooperation with Alcatel Research Belgium and emphasizes the networking aspect of the next generation interfaces. The *context of use* and the ability of the user to communicate with other users and devices in a transparent way are the central parts of this scenario. The last scenario provided in section A.3 is concerned with “ambient awareness” and social interactions. This scenario is based on a museum visit and is developed in cooperation with the Gallo-Roman Museum of Tongeren[2].

1.2 Motivation and Aims

This dissertation explores and exploits the world of Model-Based User Interface Development (MBUID) and High-Level User Interface Description Languages (HLUID) to find appropriate solutions for the problem statement introduced in section 1.1. The required interactive software necessary for the three scenarios presented in sections A.1, A.2 and A.3 in appendix A are very hard to develop with current design methodologies. There are a set of *new* requirements that can be discovered in these scenarios, that go beyond the requirements that can be detected for software targeting the desktop computer with traditional input/output hardware.

Szekely defined four challenges for Model-Based Interface Development in the introduction of the CADUI’96 proceedings [Sze96]:

Challenge 1 Task-Centered Interfaces

Challenge 2 Multi-Platform Support

Challenge 3 Interface Tailoring

Challenge 4 Multi-Modal Interfaces

This dissertation is focused on Challenge 1 and Challenge 2: creating multi-platform user interfaces starting from a task-centric design methodology. We

[2]http://www.limburg.be/galloromeinsmuseum/

believe it provides a decent solution for these two challenges based on modern tools and notations. Although Challenge 3, interface tailoring, is not explicitly addressed here, it is implicitly supported by the notation we use to describe the concrete user interfaces. Challenge 4, multi-modal Interfaces, is only examined superficially to prove it can be integrated in the framework and design process we propose in the next chapters.

Besides these four challenges, we would like to add a challenge that has become more imminent with the raise of ambient intelligence:

Challenge 5 Support for Context-Sensitive Interfaces

The necessity of this challenge is emphasized by Scenario 2 (section A.2). This challenge supersedes multi-platform support and multi-modal interfaces because they are both dependent on the context-of-use. The support for context-sensitive interfaces actually implies the next generation models should be able to anticipate dynamic changes in the user interface according to the context of use. This is not only a challenge for the notation that should be used to describe the different models, but as well as for the support that is necessary for this kind of user interfaces. This raises research challenges far beyond, but including, the Human-Computer Interaction perspective. The final chapters of this text will take the first steps to tackle this fifth challenge.

The final goal of this dissertation is to create a framework that supports the design and creation of multi-device interactive systems while sustaining a clear separation between the (user interface) designer and the (applocation logic) programmer. Multi-device interactive systems are software systems that have an important interactive part and are suitable for embedded systems and mobile computing devices. Embedded systems are computing systems that are built with "custom" hardware (according to their target environment) and usually provide less memory, processing speed and interaction capabilities (e.g. no keyboard, limited screenspace) than desktop computers. These typical constraints are addressed by our approach. An example of an embedded system can be seen in figure 1.1. Our emphasis in this disseration is to support the creation of user interfaces that can be used on multiple devices and eventually support user interfaces that can be deployed in multiple contexts of use. For this purpose we will take a task-oriented approach. Some of the challenges as proposed by Szyperski were addressed during this dissertation to reach this goal.

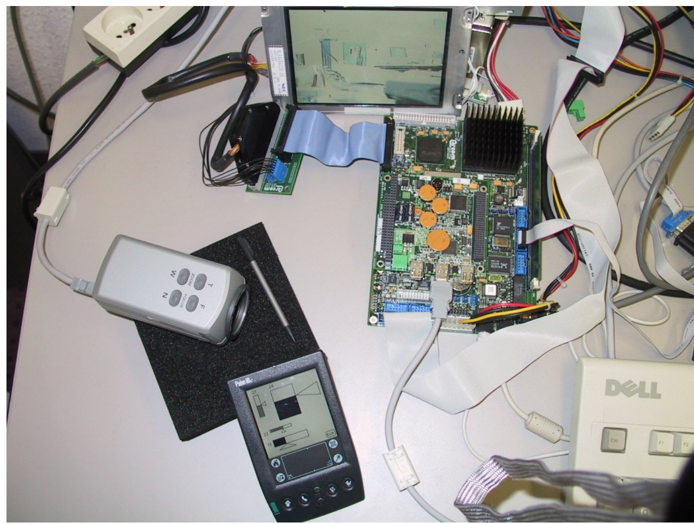


Figure 1.1: Hardware setup with Dygimes UiBuilder

1.3 Overview

This dissertation exists out of three parts:

1. Part I gives an overview of the related work and background.
2. Part II discusses our own contributions to the HCI research, design and development community.
3. Part III shows the possibilities that can be further explored based on the concepts and the framework introduced in part II.

In part I we start with a discussion of Model-Based User Interface Development in chapter 2; the definitions, history and use of Model-Based User Interface Development will be thoroughly explained. Next, chapter 3 discusses the state-of-the art of High-Level User Interface Description Language will be given, based on current approaches in XML-based user interface description languages. Both techniques have contributed significantly to multi-device user interface development.

In part II we explore our contributions that were done to the state-of-the-art in Model-Based User Interface Development and High-Level User Interface Description Language. Chapter 4 gives an overview of our framework

Dygimes. More detail about the core models that are used in Dygimes are given in chapters 5 and 6 and 8. Chapter 5 shows how the task specification is used as a central part of Dygimes. Chapter 6 introduces SEESCOA XML, an XML-based High-Level User Interface Description Languages that was targeted towards embedded systems and mobile devices. One of the application models we used was the SEESCOA component model that is presented in chapter 8. Chapter 7 provides an essential technique that is necessary to have more flexible user interface specifications: a general layout management system that can be used in an XML-based High-Level User Interface Description Language. In the last chapter of part II, chapter 9, we take a look at Uiml.net. Uiml.net is a .Net-based UIML renderer and will be compared with the Java-based approach we implemented to render the SEESCOA XML-based user interface descriptions.

In part III we look at the future of Model-Based User Interface Development and High-Level User Interface Description Languages. Since (dynamic) context starts to play an important role in user interfaces nowadays, this is something we will look into.

Chapter 2

Model-Based User Interface Development

Contents

2.1 Introduction **9**
2.2 Model-Based User Interface Development **10**
2.2.1 A Definition and some History 10
2.2.2 A More Precise Definition of Models 16
2.2.3 Our Selected Models 18
2.3 The Task Model **19**
2.4 The Dialog Model **23**
2.5 The Presentation Model **25**
2.6 Model Relations and Mappings **27**
2.7 Plasticity and Context in Models **28**
2.8 Discussion **29**

2.1 Introduction

This work builds further upon different approaches that are developed to support Model-Based User Interface Development. In this chapter we provide an overview of the important influences and state-of-the-art approaches that are of interest for the work presented in this dissertation. Both the triennial conference on "Computer-Aided Design of User Interfaces" (CADUI, [LJV04, KV02, VP99, Van96, Van93]) and the annual "Workshop on the Design, Specification and Verification of Interactive Systems" (DSV-IS, [JNF03, FLUV02, Joh01, PP00]) are meeting places for people working on Model-Based

User Interface Development and formal approaches, which are both important in our work. Both DSV-IS and CADUI are European conferences. At the ACM conference on “Intelligent User Interfaces” (IUI, [SM01, VJR04]) which is primarily held in America. At other conferences, like the “Conference on Human Factors in Computing Systems” (CHI) and the conference on “User Interface Software and Technology” (UIST) less attention is devoted to Model-Based User Interface Development as a whole, and related work presented on these conferences usually contributes to specific technologies that can be used within a model.

As discussed in chapter 1 the emphasis of this work is the support of multi-device user interfaces, more specificly on embedded systems and mobile computing devices. The revival and increased interest of the academic and industrial HCI community in Model-Based User Interface Development (and High-Level User Interface Description Languages as we will see in chapter 3) is due to the applicability of this technique for multi-device creation.

This chapter is composed as follows: section 2.2 introduces Model-Based User Interface Development. Since there has been done a tremendous amount of research on this topic since the early nineties, we limit ourselves to the related work that influenced our own approaches[1]. Next, section 2.3 highlights the task model after which sections 2.4 and 2.5 focus on the dialog and presentation model respectively. The task, dialog and presentation model are all common models used in Model-Based User Interface Development. This chapter concludes with a discussion in section 2.8.

2.2 Model-Based User Interface Development

2.2.1 A Definition and some History

Despite the great number of papers that are written about Model-Based User Interface Development, there is no clear definition what this term actually embraces. In this dissertation we speak of Model-Based User Interface *Development* instead of Model-Based User Interface *Design*. This has a particular reason: our goal is to support the whole “software engineering cycle” with the emphasis on the user interface, from the design stage up to the deployment stage. Most papers from academic literature will concentrate on the design stage of Model-Based User Interface Development, there are some however who go beyond the design. Some of them are presented in this chapter.

[1] A good starting point to get acquainted with Model-Based User Interface Development environments are the CADUI 1996 proceedings [Van96].

At the basis of Model-Based User Interface Development a *set of models* is used. A model can be informally defined as a non-empty set with elements, with a set of relations specified between these elements. A model gathers and relates information about a specific concept the final interface should reflect. A model provides an abstraction of this concept: it hides the low-level details while it preserves the important details. It typically focuses on the important characteristics that make up the interface concept; the specification of low-level details is postponed to a later stage in the design process. The elements of a model have certain values at a particular point in time. This is a very generic definition of a "model", but it allows us to think of Model-Based User Interface Development as a set of models where each model has its specific values and relations and the different models can be related to each other. These relations can be expressed in different ways: e.g. sometimes several models (partially) describe the same information or one model can be transformed into another model by adding extra information.

One of the first projects to generate a user interface by *combining* different models is Mastermind [SSC+95]; it used the presentation, application and dialog models to automatically generate the user interface [Sti97, SR98].

There are a wide range of different models that can be used in Model-Based User Interface Development: *data models*, *domain models*, *application models*, *task models*, *dialog models*, *(abstract and concrete) presentation models* and *user models* are models that are well-known and used in several Model-Based User Interface Development Environments. The data, domain and application model can be situated at the end of the application logic of the system. They define the type of objects and the operations on objects that can be used or need to be supported by the interactive system. The task and user model are closest to the user and specify the tasks the user executes and the user or user group profile(s) respectively. The dialog model and presentation model are closest to the final user interface, and will be explained in the next sections together with the task model. An emerging new kind of model is the *context model*: a model that can describe the context-of-use for an interactive system. E.g. a context model could specify a set of external parameters that can influence the appearance, usage,. . . of an interactive system. This model is the least explored, but becomes increasingly important as modern interactive systems are no longer bound to a single place and situation.

One of the first models to be used in user interface design was the *data model* or *domain model*. Its goal is to make sure the information objects used in the application will be reflected in its user interface. Information objects can be data structures or a database table for example. To overcome

the complexity of the different models and their often specific notation, tools are an important factor to make the usage of a model generally acceptable. An early tool supporting the domain model is *DON* [WD90] by Kim and Foley. What is interesting about DON in particular is its sophisticated layout mechanism. It copes with the diversity of screen-sizes ("output models") in a period where this diversity was only limited. This also means DON integrates the *presentation model* in its design methodology.

During the early nineties several groups started to create interaction models that supported multiple levels of abstraction. E.g. [PL94] introduced a toolkit that could select interactors based on the task they should accomplish. This approach has been playing a key role in the further development of model-based systems. *Trident* (Tools foR an Interactive Development EnvironmeNT) is another model-based system to create an interactive system from Bodart and Vanderdonckt [BHLV94, VB93]. It was one of the first design tools that recognized the importance of a clear separation between an abstract representation of the presentation model and a concrete representation thus supporting a multitude of interaction style alternatives for the same functional core. It also integrated task analysis as an important component to create a usable interface. Together with DON, Trident can be considered to be one of the first "complete" Model-Based User Interface Development Environments that where available.

Tadeus (Task Analysis/Design/End User Systems) is a Model-Based User Interface Development environment that focuses on practically the same models as we will do, except the domain model in Tadeus is explicitly limited to business objects [Sch96]. The Tadeus methodology uses a user model, a task model, a domain model, a dialog model and later an interaction model was added [Sch96, FS98]. Tadeus is very similar to the Dygimes approach we will present, because it also relies on automatic generation of (part of) the dialog model from the other models. On the other hand: Dygimes is more focused on multi-device user interface development, which was not the original goal of Tadeus but it was extended in a later stage by adding an interaction model [FS98] and supporting an XML-based User Interface Description Language [MFC01].

Mobi-D is a model-based integrated development environment that combines several declarative models and assists the user interface designers with the creation of these models and with the decisions she/he will have to make during the design of the user interface [Pue97]. Mobi-D offers a complete design cycle with a set of tools, and supports iterative refinements in the design of the user interface. As many others, Mobi-D works task driven: first

a preliminary outline of the user tasks is created. Next, the user-task model is created together with a domain model, and both models are closely integrated. Mobi-D emphasizes the relations between models and provides a visualization of these relations. In the following stage, a presentation model and dialog model are created in parallel. Different design tasks are supported by intelligent assistants or decision-support systems. Notice Mobi-D supports full presentation coverage, something we will define in definition 12.

Puerta makes a distinction between a Model-Based User Interface Development *system* and *environment* in [Pue97]. A Model-Based User Interface Development system may only use a limited number of selected models and does not define how these models are organized in the interface design cycle. A Model-Based User Interface Development *environment* can be grouped into three parts:

Design-time tools : tools that allow to create and relate the different models.

Runtime systems : a system that allows to execute and combine different models, resulting in a concrete user interface.

Runtime tools : tools that allow to manipulate and transform the models while executing. Notice this applies to the concrete user interface as well, because it is the result of combining the different models.

To deploy a successful Model-Based User Interface Development environment it is essential to have support for these three parts. Notice some existing approaches support all three parts, others do not. E.g. *Humanoid* [SLN92] interprets its models and generates a user interface from these models. On the other hand, *FUSE* [LS96a] generates C++ code that can be compiled into a user interface. In a Model-Based User Interface Development *environment* we assume here the models are *executable*: the environment can generate a concrete user interface from the models *without* code generation.

Figure 2.1 shows a common architecture for model-based systems and how the different models can be positioned inside Model-Based User Interface Development. Although this does not seem an architecture that puts the user central, it does actually supports user-centered design. First of all the models make sure all the user requirements are taken into account. Changing requirements implies changing one of the models, and does not imply doing a complete software engineering cycle over again. Another advantage is the possibility to have early prototypes, since the user interfaces can be generated from the different models that are included in the Model-Based User Interface

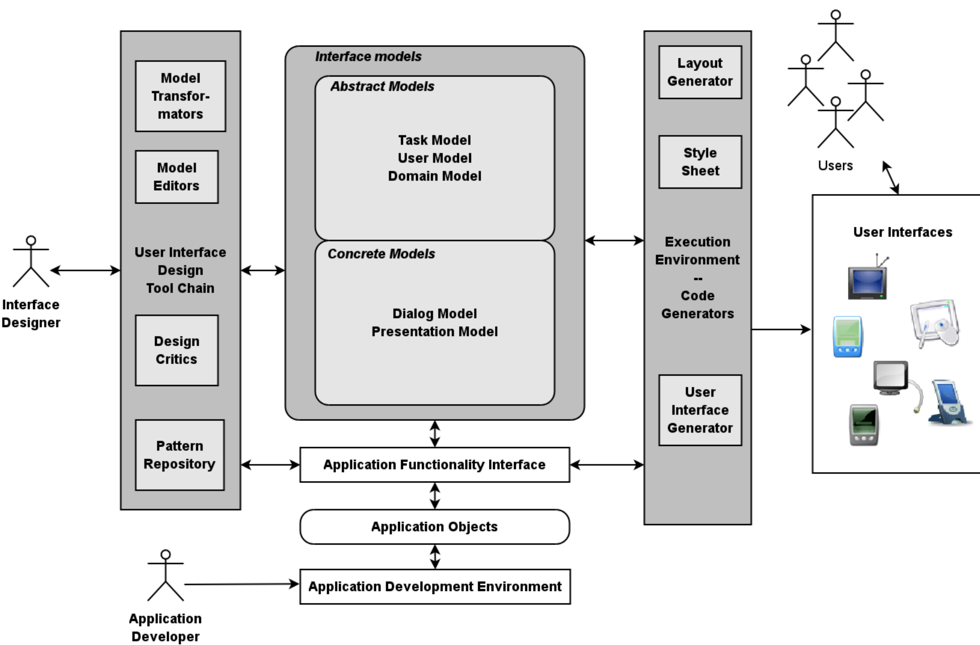


Figure 2.1: Architecture of Model-Based User Interface Development environments

Development Environment. Notice the clean separation between the software or application developer and the user interface designer. The designer can focus on what is important for the end-users, and have access to the required functionality by using the application functionality interface, while the software developer can independently develop the required application logic to drive the interactive system.

Interestingly, Model-Based User Interface Development has been reviving the last couple of years when the demand for *multi-device user interface design* was growing. By abstracting the user interface by means of (declarative) models, the creation of the concrete user interface can be postponed until the details of the target deployment platform come into play. Limbourg et. al. [LVM$^+$04a] refer to this stage as the *concrete user interface* stage.

Before Model-Based User Interface Development came into play to support multi-device user interface design, it was a mean to create user interfaces that meet the predefined requirements and that were consistent with respect to the data and functionality they needed to visualize and provide. Nevertheless,

even these early approaches included models, tools and methodologies that are suitable for multi-device user interface design and development. However, none of them succeeded as a real tool or generally accepted methodology in the industry in order to support multi-device user interface design and development. Since we witness the conception of new tools that target multi-device user interfaces, mobile interfaces and context-sensitive interfaces, there is a good chance Model-Based User Interface Development will be accepted as a new standard in user interface design. The most well-known tool is the *Teresa* tool[2] (a descendant of the ConcurTaskTrees Environment, CTTE) of Fabio Paternò et. al. [Pat00, MPS03]. This tool is built around the ConcurTask-Tree notation, which is a graphical notation for task specification. Because of the importance of this notation in the work we present here, we will discuss and use this notation in chapters 4 and 5. The Teresa and CTTE tools were both very successful, partly because of their free availability on the Internet.

We provide our own environment, **Dygimes**, that will be first presented in chapter 4, and the following chapters will highlight individual aspects of the Dygimes approach. There are several reasons why we created our own approach instead of relying on the existing approaches:

- We used a bottom-up approach in creating the Dygimes Model-Based User Interface Development framework instead of a top-down approach. We started with a concrete XML-based User Interface Description Language targeted towards embedded systems. We added support for user interface design for embedded systems by using task modeling, constraint-based layout management, dialog modeling and context-sensitive models (in that order).

- We wanted to investigate what was missing in the current approaches (as discussed before) to be accepted by the industry. For this purpose we created our own experimental setup, so we could add missing pieces easily.

- At the time this research started, there was no Model-Based User Interface Development environment available that supported a design cycle where a task model was created to serve as a basis for the creation of multi-device user interfaces using an XML-based High-Level User Interface Description Language.

[2] `http://giove.cnuce.cnr.it/teresa.html`

2.2.2 A More Precise Definition of Models

This section takes a semi-formal approach in defining Model-Based User Interface Development, for a good understanding of the rest of this dissertation. It aids in drawing up the borders of the different terms and notations that will be used throughout this dissertation. Defining a formal structure on which different models can be built has several advantages:

- assertions over the models can be checked;
- information contained in models has precise semantics;
- relations between models are unambiguous;
- and the expressive power of a model is well defined.

Nonetheless, early approaches to Model-Based User Interface Development also tried to define a (semi-)formal basis; e.g. the FUSE system [LS96a] and its related BOSS system [Bau96]. In the same conference as FUSE, Schlungbaum and Elwert discussed the use of *declarative* models [SE96b, SE96a]. Similar initiatives are still under active development in the research stage, but only some results have been accepted in industrial and/or commercial settings. Formal approaches for the development of interactive systems are not widely used outside the academic world for well-known reasons [Som04], except for specialized areas such as (hard) real-time systems and life-critical systems. There is a trend, however, to use semi-formal methods to design interactive systems to ensure the resulting product will fulfill the postulated requirements. The challenge is to provide support for rapidly evolving requirements

The previous section (2.2.1) introduced the term “model” [HC84], this section gives a more clear definition. To make assertions about models, we will need a meta-model that can be used as a framework to describe different model-based approaches and their respective properties. Let $\mathcal{M}$ be an infinite set of models.

Definition 1 A model is a tuple $< W, R_1, ..., R_k, c_1, ..., c_l >$, where W is a non-empty set over the *domain* of the model, $R_1, ..., R_k$ are binary relations defined over the members of W, and $c_1, ..., c_l$ is a set of constants of the domain W.

Definition 1 is a very general definition of a model and is therefore suitable to describe a wide range of models. Even so, we can identify a set of models that are difficult or impossible to describe in a formal way. For example, some

models are created through unstructured design knowledge, which is subjective and neither "true" nor "false" given any particular situation. A model is just a set of design information that is related with each other for one particular aspect of the interactive system. The different elements of this tuple can be explained as follows:

W contains all *elements that can capture some type of design knowledge*;

$R_1, ..., R_k$ relate different kinds of design knowledge to each other creating an *integrated pattern of design knowledge*. A binary relation R_i involves two elements of W: $R_i \subseteq W^2$;

$c_1, ..., c_l$ define the *design knowledge that is available in the model*. Since W is the domain of the model, c_j is an element of W: $c_j \in W$.

Notice this could be a partial algebraic specification of a user interface [Bau96]. Nevertheless, a pure formal notation as supported by the FUSE system [LS96a, Bau96] for example has proved to be very useful (e.g. to ensure a complete coverage of all the requirements or a correct execution of the system) but not very usable for the designer. Instead of defining a new and expressive language to describe the different models that occur in Model-Based User Interface Development by giving a more formal definition of W, $R_1, ..., R_k$ and $c_1, ..., c_l$, we will map these on existing models. The purpose of this notation is to serve as a container which can be used to compare different models and to describe the relations between the different models. We *only* introduce this notation to have a uniform definition for the concept "model". A more precise and profound formalization does not fall in the scope of this dissertation.

The concept of an integrated pattern is important here: through time we could capture reoccurring design decisions in all aspects of the design of an interactive system and we learned how to relate these different decisions to obtain a "usable" interface. Model-Based User Interface Development allows us to store this knowledge by putting it in models so their effect can persist in the final user interfaces that are generated from these models. One initiative that has successfully created a general approach towards Model-Based User Interface Development is presented in [LV04] by Limbourg and Vanderdonckt: graph structures are used to have a general description of a model and its contents while graph transformations can describe the relations between the different models or model iterations.

In our approach we will only focus on a limited subset of models of $\mathcal{M}$: these selected models constitute a Model-Based User Interface Development

environment that will be denoted by $\mathcal{E}$. For the remainder we fix an environment E that uses a set of models $\{M_1, M_2, ..., M_n\}$. Notice we work with an *environment* instead of a system as defined in [Pue97]. This notation gives rise to definitions 2 and 3.

Definition 2 By $\mathcal{M}$ we denote an infinite set of models.

A model-based system is a piece of software that uses a set of models to support the design of user interfaces. Examples of model-based systems that encompass several different models are Trident, Tadeus, Mobi-D, Teresa and Dygimes.

Definition 3 By $\mathcal{E}$ we denote the mapping that assigns to every system S a set of models $\mathcal{E}(S)$. We call $\mathcal{E}(S)$ the environment for S, and $\mathcal{E} : S \rightarrow 2^{\mathcal{M}}$.

2.2.3 Our Selected Models

The work we present here focuses on three specific models that can support the design of a whole user interface. These models have proved to be sufficient w.r.t the design of different kinds of user interfaces e.g. speech interfaces, form-based or web-based interfaces, 3D interfaces,... The three models that will serve as a basis for this dissertation are:

***Task* Model** ($\mathcal{M}_{\mathcal{T}}$)

***Dialog* Model** ($\mathcal{M}_{\mathcal{D}}$)

***Presentation* Model** ($\mathcal{M}_{\mathcal{P}}$)

We will consider these three models as the *core models* in our Model-Based User Interface Development approach. There are also the application and domain model that are available in our system, but its sole purpose is to bind the user interface and application logic together, so we don't consider this to be a core model in Dygimes. Instead, one of the goals is to have a clear separation between the user interface and the remainder of the interactive system, so we aim at supporting multiple application/domain models. The core models can be supplemented with other models that are necessary given a particular context. For example, when designing the interactive system with a particular focus group in mind a user model needs to be added to the models to capture all these requirements. The environment Dygimes that will be introduced in chapter 4 can now be defined as a triple in the notation introduced in

section 2.2.2: $\{\mathcal{M}_\mathcal{T}, \mathcal{M}_\mathcal{D}, \mathcal{M}_\mathcal{P}\}$. According to definition 3 $\mathcal{E}(Dygimes) = \{\mathcal{M}_\mathcal{T}, \mathcal{M}_\mathcal{D}, \mathcal{M}_\mathcal{P}\}$.

In general, the different models can be divided into two separate classes: abstract models and concrete models. The former expresses information that has no direct concrete counterpart in the final user interface, while the latter can be easily mapped on elements of the final user interface. Examples of abstract models are the task model and domain model, while examples of concrete models are the dialog model and presentation model.

One of the aspects that makes this work different from other approaches is the *multi-device run-time* aspect. Instead of using these models merely in the design phase, they will also be used *at run-time*. We say this collection of models become *executable* in multiple environments. Given an interpreter or specialized environment for execution the models do not need to be converted into program code before they can be used in applications, but are considered part of the runable application. Executable models are not new, but making models executable on multiple target devices is a new research topic. An introduction to the different selected models is given in sections 2.3, 2.4 and 2.5.

2.3 The Task Model

Within task analysis a designer creates task specifications to describe the activities necessary to reach a goal. There is no agreement for an exact definition of task, activity or goal. Intuitively: a *goal* is the result the user wants to obtain after a (set of) *task*(s) is/are performed. Each task can be accomplished by executing a set of subtasks or *actions*. Subtasks are just tasks that have more detail for a part of their parent task. Executing all subtasks of a task, results in the execution of the task itself. An action can not be divided in sub-parts: it is an atomic operation that is executed upon an artifact, by an entity that is involved in the completion of the task (user, computer,...). To illustrate a task model, consider figure 2.2 where a simple task specification is depicted for getting a drink from a vending-machine. The goal for this task specification is to satisfy your thirst. The top level task is to “get a soft drink” from a vending machine. This task is further divided into subtasks. Notice these subtask could also be subdivided further, according to the granularity the designer wants to use to specify the task.

Task specifications are based on a notation provided by a task model. We define a task model in definition 4. A task model will be redefined more formally by definition 13 in chapter 5.

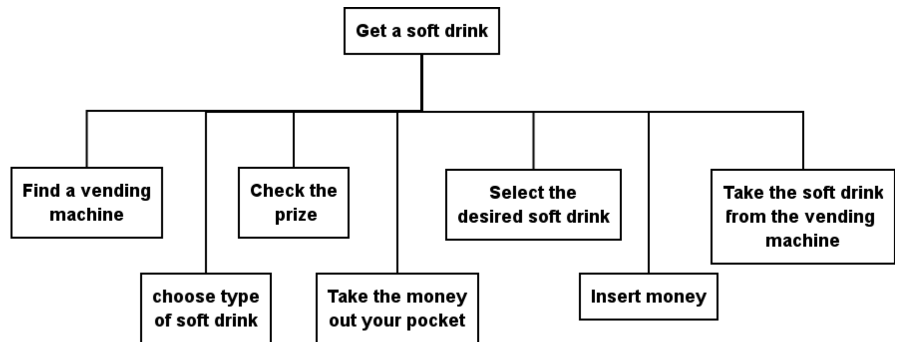


Figure 2.2: An example task specification for getting a soft drink.

Definition 4 A *task model* $\mathcal{M}_{\mathcal{T}}$ is a notation to describe the activities, tasks and subtasks that are performed to reach an arbitrary goal and the relations between them. A task model offers a way to structure and represent information about activities, tasks and subtasks and serves as a template for the result of task analysis: task specifications. A task specification $T_i \in \mathcal{M}_{\mathcal{T}}$ is the definition of a presentation using the structure and notation defined in $\mathcal{M}_{\mathcal{T}}$.

Let $\mathcal{T}$ be an infinite set of tasks $\{t_1, ...\}$. By $\mathcal{R}$ we denote the set of possible relations between tasks. If we are dealing with an hierarchical task specification, there is always the decomposition relation that exist in $\mathcal{R}$. The decomposition relation d relates a *single* task t to a set of *sub*tasks $\{t_1, ..., t_n\}_{n \geq 2}$: this relation could be expressed graphically as: $t \xrightarrow{d} \{t_1, ..., t_n\}_{n \geq 2}$. If t is a leaf task and can not be decomposed any further than $t \xrightarrow{d} \emptyset$. We will see there are other relations between tasks, depending on the task specification notation that is being used. The next paragraph will show which relations the ConcurTaskTree notations adds to $\mathcal{R}$.

As a notation for task modeling we choose the ConcurTaskTree notation, since in our opinion it is the most usable and modern ("context"-ready) specification notation. The ConcurTaskTrees task model is a notation proposed by Fabio Paternò [Pat00] for designing a task specification. This notation offers a graphical syntax, an hierarchical structure and a notation to specify the temporal relation between tasks. Four types of tasks are supported in the ConcurTaskTrees notation: abstract tasks, interaction tasks, user tasks and application tasks. These tasks can be specified to be executed

in several iterations. Sibling tasks, appearing in the same level in the hierarchy of decomposition, can be connected by temporal operators like choice (`[]`), independent concurrency (`|||`), concurrency with information exchange (`|[]|`), disabling (`[>`) , enabling (`>>`), enabling with information exchange (`[]>>`), suspend/resume (`|>`) and order independence (`|=|`). [PS02] specifies the following priority order among the temporal operators: *choice* > *parallel composition* > *disabling* > *enabling*. Notice we have defined the set of temporal operators $\mathcal{R}$ for the ConcurTaskTrees notation now: $\mathcal{R}$ = {`[]`, `|[]|`, `|=|`, `[>`, `>>`, `[]>>`, `|>`, `|||`} $\cup\{d\}$ where d is the decomposition relation.

Parts of the ConcurTaskTrees notation are based on the LOTOS notation [LFHH91], one of the products the Open Systems Interconnection (OSI) standardization products. LOTOS is a standard that is maintained by the International Organization for Standardization (ISO). Four properties of the LOTOS language come in handy to use them in a notation for task modeling:

Temporal operators : Temporal relations between different processes can be specified in a convenient way.

Support for concurrency : Processes can execute concurrently (a situation that is often the case in distributed systems).

A well-defined process algebra : Statements over a LOTOS specification can be proved true or false.

Executability : A LOTOS specification is *executable*; it can be executed by an interpreter for example.

[Pat97] shows how LOTOS specifications can be reasoned upon by transforming them in Action-Based Temporal Logic formulas. Unfortunately some formalism that is offered by the LOTOS notation is no longer available in the ConcurTaskTrees notation. The main reason is the purpose of the ConcurTaskTrees notation: to enable a human user to design a task specification for common tasks. In its current state, the ConcurTaskTrees notation could be classified as a *semi-formal* notation. There are only a few attempts to recover the formal basis of the notation and use it to validate, emulate and process the task specifications. In chapter 5 we will create a semi-formal basis to reason with ConcurTaskTrees task specifications. One of the main objectives is to find an algorithm to detect a correct set of enabled task sets and to create a dialog model from the ConcurTaskTrees task model.

One notation that proceeded ConcurTaskTrees was the eXtended User Agent Notation (XUAN) notation [GEM94], an extension of the UAN notation [HSH90]. XUAN provides a notation that can be considered equally

expressive as the ConcurTaskTrees notation, but does not provide a graphical syntax. This makes the language less suitable for designers. UAN was probably one of the first task modeling languages that introduced temporal relations in the task model. These temporal relations are based on Allen's temporal logic[All84]. Before UAN the focus was on (a) measuring or predicting execution time of tasks (e.g. GOMS) and (b) applying guidelines w.r.t. execution time of tasks. UAN, and later on XUAN, were the first that allowed to express temporal properties of interaction in the task specification. There are many resemblances between the XUAN notation and the ConcurTaskTrees notation, except the ConcurTaskTrees notation has a graphical syntax. For example, UAN defines five possible temporal constraints between tasks: "sequence", "order independence", "interruptible by", "interleavable with" and "can be concurrent with". XUAN extends this notation with parameter passing between tasks, post- and pre-conditions. A next generation of UAN exists: "Pattern User Action Notation" (PUAN)[ED04]: it integrates patterns and the UAN notation to support multi-platform task modeling.

Since the ConcurTaskTrees notation allows concurrent tasks, this implies several tasks can be valid in the same period of time. We can identify sets of tasks (tasks being leafs in a ConcurTaskTrees specification) that are valid during the same period of time by inspecting the temporal operators between the different tasks. This is called an **Enabled Task Set** and is defined in [Pat00] and repeated here in definition 5.

Definition 5 An *Enabled Task Set* (ETS) is a set of tasks that are logically enabled to start their performance during the same period of time. The enabled task set of a given task specification $T \in \mathcal{M}_{\mathcal{T}}$ is denoted by $E(T)$ where $E : T \rightarrow 2^{\mathcal{T}}$.

Notice a task can belong to several enabled task sets according to this definition, which is normal since the specification allows concurrent tasks. Finally, definition 6 introduces an *Enabled Task Collection* based on the definition of an enabled task set.

Definition 6 An *Enabled Task Collection* (ETC) E is a set of sets of tasks $E \subseteq 2^{\mathcal{T}}$.

Notice a task model can be described using definition 1, the ConcurTaskTrees notation fits nicely in this definition. In $< W, R_1, ..., R_k, c_1, ..., c_l >$, W is exactly the set of possible tasks that can be used in a task specification (the "domain"). $R_1, ..., R_k$ are all the relations that can connect two elements of

W: In ConcurTaskTrees {`[]`, `|[]|`, `|=|`, `[>`, `>>`, `[]>>`, `|>`, `|||` , d} (where d is the decomposition relation) are the types of relations that can be used.

2.4 The Dialog Model

There are different possible interpretations for what a dialog model can represent. One of the reasons is that the dialog model is probably the model that is the least explored and the hardest to edit [Ols92]. Four important milestones can be identified in the quest for the meaning of "dialog" in user interface development:

1. In the first stage, one tried to *understand a dialog*, its properties, and concepts. A dialog should describe the interaction between a user and a user interface.

2. The second stage involved *modeling a dialog* which remains an open question. Gilbert Cockton gives the advantages and disadvantages of five dialog models in [Coc87] (Backus-Naur-Form grammars, state transition networks, production rules, Hoare's Communicating Sequential Processes (CSP), and Petri nets). These five dialog models are compared leading to a conclusion that none of them holds all the desired properties. Green [Gre86] reported that event/responses languages are more expressive than grammars and state transition networks.

3. The third stage involved *acquiring design knowledge* for producing a quality dialog from existing sources of information: for instance, expressiveness, parametrized modularization, and executability are properties of interest that should be captured in design knowledge.

4. The last stage involved *generating dialog* by incorporating part of this design knowledge and by relying on modeling concepts: dialog is definitely governed by task configuration, although dialog and presentation usually work hand in hand.

Since there are different possible notations for a dialog model, we illustrate it with the notation we will use in later chapters. Figure 2.3 (from [DFAB04]) shows an example state transition network that describes the behavior of part of an editor. The transitions between states are labeled by the keyboard keys that trigger the transitions.

In the last and fourth stage we have enough knowledge to automatically generate a dialog (model), partially based on information present in the task

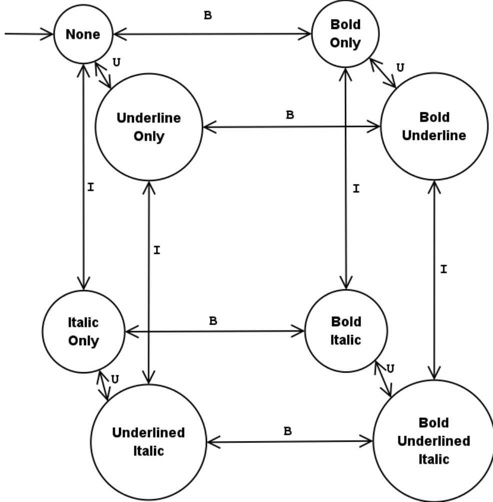


Figure 2.3: An example dialog specification from [DFAB04]: a state transition network that describes the user actions to change the text style

model (section 2.3). Nevertheless, the *exact* definition of dialog depends on the notation that is used to represent the dialog model (what information can be represented). In this dissertation, we define a dialog model in definition 7 based on the definitions we provide for task model in section 2.3.

Definition 7 A *dialog model* $\mathcal{M}_{\mathcal{D}}$ is a notation to describe the relations that exists between the set of tasks that are valid at one point in time and the presentation units that represent this set of tasks. A dialog specification $D \in \mathcal{M}_{\mathcal{D}}$ is the definition of a presentation using the structure and notation defined in $\mathcal{M}_{\mathcal{D}}$.

We consider the dialog model as an intermediate model between the task model and presentation model. Unlike the task model we use, it is closer to flowchart modeling for example, but besides information flow it also reflects how navigation between the set of tasks is executed. Unlike the presentation model it includes relations between tasks and presentation units (or between dialog screens, see section 2.5) but does not specify the internal structure of the dialog. The representation of a dialog is the set of presentation units that is related to the dialog.

It can be argued the dialog model is another view on the task model with this definition. This is partially true: in our approach we will generate a dialog

model that is based on the task model by grouping the tasks and using the temporal relations from the task model. On the other hand, in contrast with the task model, the dialog model specifies the progress throughout the user interface, a starting and ending point and the events that drive the progress. The temporal and decomposition relations will no longer be visible in the dialog model, while transitions between task sets covering navigation will become available.

We say a dialog specification offers *full dialog coverage* if all the tasks of a task specification are *reachable* in the dialog specification. Thus there is always a set of navigation events by which a leaf task in the task specification can be executed. A navigation event that changes the state of the dialog specification is usually triggered by the execution of a task. Definition 8 introduces full dialog coverage. Since we will present the dialog specification as a state transition diagram, in which each state is a set of tasks that are "active" during the same time period, we can use the underlying structure of a directed graph to define full dialog coverage.

Definition 8 A dialog specification D offers *full dialog coverage* for a task specification T if and only if $\forall$ leaf task $t_f \in T$, $\exists$ a state s_n with $t_f \in s_n$ and a directed path $\{s_t, s_{t+1}, ...s_{t+i}, ..., s_n\}, 0 \leq i \leq n$ where s_t is a state with no incoming edge.

Referring back to definition 1, a dialog model can also be expressed as the tuple $< W, R_1, ..., R_k, c_1, ..., c_l >$, W are all the possible states in a dialog model. $R_1, ..., R_k$ are the relations between two states: a relation is a transition from one state to another.

2.5 The Presentation Model

The presentation is the most concrete realization of the user interface: on a desktop system the presentation could be a window and its widgets that are shown, on a mobile phone the presentation could be presented as a voice interface. The presentation of a user interface is the physical manifestation of the available means to communicate with a (software) system. For a clear understanding definition 9 gives an informal definition of *presentation unit*.

Definition 9 A *presentation unit* u groups the concrete realization of the interface(s) (or building blocks) that can be manipulated by the user(s) in a certain well-defined period of time.

A presentation unit is typically a set of *Abstract Interaction Objects* (AIOs) [VB93] that will be mapped on a set of corresponding *Concrete Interaction Objects* (CIOs) to create the final user interface that will be presented to the user. AIOs are abstract representation of interface objects that can interact with the human user, and CIOs are the concrete representation; e.g. a "range indicator" is an AIO and can be mapped to a slider widget (CIO). A CIO is the "physical" realization of an AIO depending on the modality that is chosen: e.g. it can be the graphical representation of widget or a part of a spoken dialog. In [VB93] a set of selection rules (structured as a selection tree) is defined to map AIOs on the appropriate CIOs in Trident.

Others have defined presentation unit in a slightly different way. Eisenstein et. al. define a presentation unit in [EVP01] as a composition of one or more logical windows in which a logical window groups AIOs, based on the definition of a presentation unit given in [BHL+95]. A presentation unit is related to a sub-task of the user task in this definition. Our definition expresses the same concept, but emphasizes a presentation unit is related to the concept of a dialog instead of a task. A presentation unit is related to the user task by introducing user interface *building blocks* (see definition 10).

Definition 10 A *building block* is a set of Abstract Interaction Objects that represent a leaf task from a hierarchic task specification.

We define a presentation model in definition 11.

Definition 11 A presentation model $\mathcal{M}_{\mathcal{P}}$ is a notation to describe the set of presentation units (def. 9) that occur during the lifetime of a application. A presentation specification $P \in \mathcal{M}_{\mathcal{P}}$ is the definition of a presentation using the structure and notation defined in $\mathcal{M}_{\mathcal{P}}$.

Notice definition 9 can also be applied to distributed user interfaces, and interfaces that use different modalities. A presentation unit is not limited to one interaction technique; e.g. it can make use of different interaction devices simultaneously [VC04]. The definition of a presentation unit can be related easily with the enabled task set definition in section 2.3: there is a one-to-one relation between a presentation unit and an enabled task set. For each enabled task set a presentation unit should be provided: if this condition is fulfilled the designer is certain the user interface will cover every aspect specified in the task specification. This is called a *full presentation coverage* of the task specification by the presentation model as defined in definition 12.

Definition 12 A presentation specification P offers *full presentation coverage* for a task specification T if and only if for each leaf task $t_f \in T$; $\exists$ a presentation unit u where t is presented by a part of u.

Full coverage for a task specification indicates the final user interface covers all functional requirements: all functionality that should be exposed or should be accessible by the user according to the requirements is available through the user interface.

Again, referring back to definition 1, a presentation model can be fit in this definition. The domain W of the model is the set of AIOs and CIOs, and the relations $R_1, ..., R_k$ are al the possible relations between AIOs mutually, CIOs mutually and between AIOs and CIOs like mapping, decomposition, positioning w.r.t. each other,. . . .

2.6 Model Relations and Mappings

So we have a set of models that make up our enviroment, where we still have to define inter-model relationships, where one model is related to another model in a particular way. These inter-model relationships can be classified as the different variations of the *mapping problem* [PE99]. A mapping can be interpreted as a function that takes a model as input and gives a model as output: $Map :< W, R_1, ..., R_k, c_1, ..., c_l > \rightarrow < W, R_1, ..., R_k, c_1, ..., c_l >$. The Map function can only rely on the input model to generate the output model, or require parametrization. The former would support automatic transformations that do not need extra information to execute the transformation, the latter could be part of an interactive design environment where the designer needs to input extra information to execute the transformation.

In [CLC04d] we identified two mechanisms to solve the mapping problem. In addition to the three mechanisms that were already introduced by Limbourg et. al. in [LVS00] we get the following five possible inter-model relations:

1. **model derivation**: constructing an unspecified model using the information of an already specified model. For example: deriving a dialog model from a task model [LVS00, VLF03, MPS03, LCCV03].

2. **partial model derivation**: elements of a model or relationships between elements are added to a model. For example: adding transitions between states of a dialog model while examining a task model [LCCV03].

3. **model linking**: connecting distinct models to each other. For example: linking presentation units to unit tasks of a task model [CLV+03].

4. **model manipulation**: the human designer applies changes to a model. For example: manually completing a skeleton of a model [Pue97].

5. **model update**: updating a model as a result of changing or adding properties in another model by the human designer or an algorithm. For example: updating the task model when parts of the presentation model are changed [Sti99].

These different mechanisms are exactly the relations between the different tuples in $\mathcal{E}(S)$ where each tuple represents a different model.

2.7 Plasticity and Context in Models

Each presentation unit can have a certain degree of *plasticity* [The01]: a scale of the adaptability of the user interface within a single presentation unit. Thevenin, Calvary and Coutaz defined a framework for using plasticity in the development of interactive systems [TC99, CCT00, CCT01]. Plasticity as it is defined here is clearly focused on the presentation model, and has only little relations with other models. If we consider the research work that has been done to create context-sensitive user interfaces, the notion of plasticity should be introduced in several other models used in Model-Based User Interface Development. The models which can change during the lifetime of the application because of influences of the context are called *dynamic models*. We identified the dialog and presentation model are all dynamic models when the target is a single-user context-sensitive interactive system. The task model is a semi-dynamic model, because even in context-sensitive interactive systems the goal will remain the same irrespective of any context change, but the sub-tasks and structure of the task model could change according to the change in context. Although other dynamic models could be identified we limit the discussion to these three models; the user model and domain model are considered static models.

In addition to the definition of plasticity we introduce the concept of *dialog plasticity* in a user interface. This plasticity applies to the navigation between dialogs and dialog structure in the user interface. Consider the different models that can be influenced by a context change: all dynamic and semi-dynamic models can change *at run-time* because of a context change. Since these models are probably related to each other, the change in one model will be

propagated to the other related models. In particular, a change in the task model will influence the dialog model thus changing the navigation and structure of the user interface. If this happens, the dialog model will be changed at run-time according to the new context, but this change should be limited to ensure the usability of the user interface. Exactly this limited change of the dialog model is called dialog plasticity. Chapter 10 discusses our attempts to create a context-sensitive task model and shows how this influences the dialog model. Some preliminary work we published on this topic of context-aware Model-Based User Interface Development can be found in [CLC04b] and [CLC04a]

2.8 Discussion

There is a clear shift from traditional user interfaces (e.g. the ones used for desktop computers) towards "pervasive", "ubiquitous" or "context-sensitive" user interfaces. Model-Based User Interface Development provides a framework that is flexible enough to cope with these new kinds of interfaces. The interface can be abstracted into a set of models for describing the properties of an interactive system. The conception of the final, concrete presentation of the interactive system is postponed as much as possible to gain flexibility and reusability of user interface design.

In this chapter, we provided an overview of the foundations of Model-Based User Interface Development, and discussed three models from Model-Based User Interface Development in particular. The semi-formal notation was solely introduced to have a clear understanding of the concepts that are being used throughout this dissertation. The task model allows to integrate the task analysis in Model-Based User Interface Development. The dialog model can be considered as a transformed task model or, depending on the granularity of the task model, as a flow of "interaction sessions" reflected in separate presentation units. There are two important properties a Model-Based User Interface Development environment should satisfy: it should provide *full dialog coverage* and *full presentation coverage* for a task specification.

Chapter 3

High-Level User Interface Description Languages

Contents

3.1 Introduction 31
3.2 History of XML-based User Interface Description Languages 33
3.3 An overview of XML-based High-Level User Interface Description Languages 35
3.3.1 Abstract User Interface Markup Language (AUIML) 35
3.3.2 Renderer Independent Markup Language (RIML) . . 36
3.3.3 Useware Markup Language (useML) 37
3.3.4 Teresa XML 38
3.3.5 Interface Specification Meta-Language (ISML) . . . 39
3.3.6 The User Interface Markup Language (UIML) . . . 40
3.3.7 XIML 41
3.3.8 UsiXML 42
3.4 Discussion 43

3.1 Introduction

In this chapter we will provide an overview of High-Level User Interface Description Languages, more specific XML-based High-Level User Interface Description Languages. A separate chapter is devoted to this topic because of the importance of XML-based High-Level User Interface Description Languages for multi-device user interface development. The most important benefit this

technique provides is the possibility to create a *device-independent* and abstract description of a user interface with a language that is easy to use in heterogeneous environments. Although there are also many User Interface Description Languages that do not use XML, we feel XML is the most appropriate language to describe High-Level User Interface Description Languages. Chapter 6 will give a more complete motiviation in section 6.3.

Other overviews of the state-of-the-art in High-Level User Interface Description Languages can be found in [LALV04], [The01] and [SV03]. [SV03] compares different XML-based High-Level User Interface Description Languages on different aspects: models, methodology, tools, supported languages, supported platforms, target, level, tags, expressibility, openness and concepts. In [The01] the focus is on plasticity of user interfaces and the fact that different XML-based User Interface Description Languages are discussed here among other approaches provides evidence for the applicability of XML-based User Interface Description Languages for multi-device interface development.

The richness of the user interfaces is proportional with the expressive power of the presentation model. There are two extremes w.r.t. the expressive power: the *common denominator* and the *meta-widget set* approach. On the one hand the common denominator approach identifies a general set of widgets that can be used on most platforms or devices. On the other hand, the meta-widget set approach avoids to include all widget set specific information from the presentation model. To progress from the XML-description to the concrete widgets in the user interface a mapping phase is necessary. In this chapter we will focus on the models where the different languages can be situated, how they provide linking with different models and the complexity and expressive power they provide. Chapter 6 will introduce SEESCOA XML, a custom XML-based User Interface Description Language we created in early 2001 for embedded systems and mobile computing devices.

During the workshop on *Developing User Interfaces with XML: Advances on User Interface Description Languages*[1] in Gallipoli [LALV04] an overview of the state of the art of XML-based High-Level User Interface Description Language in 2004 was provided. Most of this chapter is based on the various papers that were presented on this workshop. After tracking down how and when the first conception of XML-based High-Level User Interface Description Languages happened in section 3.2, we will provide an overview of the different XML-based High-Level User Interface Description Languages in section 3.3:

- AUIML (section 3.3.1)

[1]http://www.edm.luc.ac.be/uixml2004

- RIML (section 3.3.2)
- Teresa (section 3.3.4)
- useML (section 3.3.3)
- ISML (section 3.3.5)
- UIML (section 3.3.6)
- XIML (section 3.3.7)
- Usixml (section 3.3.8)

This list is not exhaustive, and there are many more existing languages. The purpose of this chapter is not to have a complete comparison of the different languages, it provides an overview to illustrate how XML is becoming *the lingua franca* for user interface descriptions. The reason why a comparison can not be the main goal of this chapter is the different levels of abstraction on which the discussed XML-based High-Level User Interface Description Languages operate. Instead we try to identify in which models from traditional Model-Based User Interface Development the notations can be used and an overview of our findings is provided in tables 3.1 and 3.2.

3.2 History of XML-based User Interface Description Languages

For about a decade there exist *declarative* User Interface Description Languages, e.g. [SSC+95],[Sti97] and [Pin00]. Declarative meaning the designer can specify *what* she/he wants instead of *how* to create the envisioned result. Besides providing researchers with a solid foundation to develop other concepts these declarative user interface specifications could not be set free from academic research. Although there have been real-life uses of these approaches and everyone agrees they result in better user interfaces, they did not influence the current technologies for building user interfaces. One of the reasons being the specialized languages the designers or developers need to master to use these declarative models.

During the nineties the popularity of the world wide web was responsible for the education and training of many interface designer that were necessary to create webpages, but had virtually no experience with "traditional user interfaces". This proved a markup language is much more comprehensible

for these designers to create interfaces then a programming language is. Most website designers have no programming background but are still able to create an interactive system. Since the omnipresent nature of the world wide web as an information resource lead the first successful adoption of real device independent interface creation. From the W3C Device Independence Workgroup website (`http://www.w3.org/2001/di/`):

> *the number of different kinds of device that can access the Web has grown from a small number with essentially the same core capabilities to many hundreds with a wide variety of different capabilities. At the time of writing, mobile phones, smart phones, personal digital assistants, interactive television systems, voice response systems, kiosks and even certain domestic appliances can all access the Web.*

Long before it became popular in traditional user interfaces, a webdesigner could separate content [con01c] and style [con01a] and make the content presentable on every kind of device that has a graphical output capability. As a consequence the most usable specifications to store and communicate device profiles can be found in the web development world: e.g. UAprofile[2], and more general CC/PP [con03].

An indication of the importance of the web-development approach where markup is used to create (multi-device) user interfaces, was the introduction of *Xaml*[3] by Microsoft; a user interface oriented markup language that is suitable to create "fat" (desktop) clients with the background knowledge of a webdesigner. Just like HTML can be tight to the Document Object Model (DOM, [con01b]), Xaml integrates smoothly with the Microsoft Windows desktop API.

XForms [con01f] is another example where the same kind of evolution can be observed: primarily targeted towards web-based forms, it is also used to implement user interfaces on multiple devices. XForms also offers a degree of abstraction comparable to the other approaches presented in the next sections and has support for a web-based domain model [VLC04]. The XML User interface Language (XUL) [HGHW01] is another XML-based User Interface Description Language we should not forget: initially it was a language that allowed to create user interfaces inside the Mozilla[4] browser. It takes advantage of other languages that are supported in the browser like HTML, Cascading

[2] `http://w3development.de/rdf/uaprof_repository/`

[3] `http://longhorn.msdn.microsoft.com/lhsdk/core/overviews/about%20xaml.aspx`

[4] `http://www.mozilla.org`

StyleSheets (CSS), Resource Description Framework (RDF) and JavaScript. There are several XUL renderers that can be used outside the browser and support part of the XUL specification.

Observing this evolution from a model-based point of view, changes to the type of application model that is supported can be detected. Traditional application models where based on an entity-relationship model and, afterward, on the object-oriented design models. The latter is still the case for several XML-based User Interface Description Languages discussed in this chapter. With the current merge between fat clients and thin clients by means of universal XML-based User Interface Description Languages, the boundaries to use web services as application model are disappearing. In our own approach we support an open ended specification so different kinds of application models can be used if the protocol to communicate with the application objects is supported. This will be further explained in sections 4.5 and 6.4.

3.3 An overview of XML-based High-Level User Interface Description Languages

3.3.1 Abstract User Interface Markup Language (AUIML)

The Abstract User Interface Markup Language [MWK04] was conceived at IBM and is available at the IBM alphaworks website (`http://www.alphaworks.ibm.com/tech/auiml`). It is based on the Panel Definition Markup Language (PDML); an XML-language that was originally created to build portable Java Swing applications. AUIML targets form-based user interfaces and uses an *intent-based approach*. An example of an intention is a choice that has to be made by the user: AUIML allows to specify this intention rather than using GUI specific identifiers. It is only very recently IBM has released AUIML and a related tool as a plugin for the Eclipse software development environment. Tool support is essential for AUIML, since it is not a human readable syntax that is being used.

AUIML tries to avoid the common denominator problem by offering a property mechanism the designer can use to leverage the device-specific presentation capabilities. This is not reflected in a schema for the language however and requires rather complex mapping mechanisms. This property mechanism uses a description similar to Cascading Stylesheets [con01a] and uses a separate *properties* file. This does not make the XML code much more readable: what is presented by the AUIML document is only comprehensible in conjunction with the application code that is related with the user interface description.

AUIML is clearly focused on the *presentation model*, and does not have a strong commitment towards involving other models than the *data model*. The XML language is data-driven: it defines the type of information that has to be presented as a tag, and provides a mapping for each type that is used in the AUIML document. Each type also has a binding with the data model, in this case a (set of) Java object(s).

To summarize, AUIML provides a separation between the user interface structure and style (the properties file). It allows to map abstract interactors on concrete widgets in Java Swing or HTML. It has a grid-based layout mechanism: each abstract interactor allocates a cell in a predefined grid. An AUIML user interface description also has a one-on-one mapping with a set of application objects. AUIML supports the different aspects that are typical for a presentation model: the *structure* of the user interface, a *layout* description, *rendering hints* that describe CSS-like style properties and *widget mappings* that map AIOs on CIOs. These four properties are defined more thoroughly in section 6.1.

3.3.2 Renderer Independent Markup Language (RIML)

The Renderer Independent Markup Language [KWWZ04] is conceived as part of the Consensus project (`http://www.consensus-online.org/`) and based on XHTML 2.0 and XForms 1.0. According to the Consensus website:

> *RIML allows an author to write content only once in a standardized format based on the Extensible Hypertext Markup Language (XHTML) and XForms. The content is then automatically adapted for different end-user terminals including voice.*

RIML is clearly focused on form-based user interfaces and adds speech-support to these interfaces. It relies heavily on a web architecture: transformation is done server-side in an environment with sufficient computing power.

RIML focuses on *layout*, *pagination*, *navigation*, *voice support* and *content selection*. The layout mechanism of RIML is rather simple but effective: it allows to lay out the interface using rows, columns and grids similar to but a little more powerful as AUIML. The innermost container in a RIML document is called a “frame” and is the only element that can have actual content. Pagination is the core layout mechanism to have a scalable interface that “fits” for different devices.

To summarize, RIML is heavily based on XForms for the data model and event handling that are being used. XHTML defines the structure of the presentation and it provides a RIML-specific layout description which is separated

from the content and structure. RIML builds further upon the early work that has been done to make websites available on different browser-platforms. By doing this it inherits the abstractions for user interface design that were already available in the existing markup languages (XForms, XHTML). Merging (part of) the different markup languages also results in a fairly big specification. However, RIML adds some new concepts like intelligent pagination and allows to specify alternative content for the different output channels within the interface description. RIML supports three of the different aspects that are typical for a presentation model: structure, layout, and rendering hints. Widget mappings are not included since RIML uses standard markup languages.

3.3.3 Useware Markup Language (useML)

The Useware Markup Language [ZMBR04] is a fairly new XML-based User Interface Description Language based on "Useware" (`http://www.uni-kl.de/pak/useML/`). The Useware development process considers different users, their tasks and their experiences. This language is included because it offers another viewpoint: instead of traditional AIO - CIO mappings, useML provides an abstraction of a user interface by "Use Objects" (UOs) and "Elementary Use Objects" (EUOs). These types of objects also provide the structure of the task: UOs define the tasks and can be hierarchically structured, and EUOs define the actions of this task. UseML is a user-centered interface development language with a very specific domain and coverage. The domain is the operation of machines, although this could be generalized to other domains. UseML covers "structuring" of the user interface and helps in the analysis and design of the user interface. For the presentation (or realization) of the user interface other languages and tool should be used.

It is clear useML is closer to the task model than it is to the presentation model. Use Objects can contain five elementary task types: execute a function, select a value, give data input, change existing data and inform the user (give data output). Each type has a precise description in its schema but it is not defined how the task types are made concrete in the final user interface since this is done with other means (e.g. UIML).

To summarize, useML is a high-level User Interface Description Language that focuses only on structuring the user interface from a task analysis point of view. For the creation of the final user interfaces the designer should use other means such as XHTML, UIML or similar User Interface Description Languages that do cover a presentation model. There is no clear binding

with an application domain so we assume this is generated and controlled by the transformation process. UseML supports one of the different aspects that are typical for a presentation model: it does recognize the importance of structuring the user interface. The other three aspects (layout, widget mappings and rendering hints) are not supported in the specification.

3.3.4 Teresa XML

Teresa XML [BCPS04] is the XML-based High-Level User Interface Description Language that is used together with the ConcurTaskTrees notation in the Teresa environment (`http://giove.cnuce.cnr.it/teresa.html`). Since Teresa XML combines an XML-based High-Level User Interface Description Language with the XML-based notation to describe the task model, this notation supports the *task model* as well as the *presentation model*. Similar to the approach we will present in the next chapters, the dialog model is derived from the task model. The dialog model is made “explicit” by connecting the Teresa XML abstract user interface description with the description of the ConcurTaskTrees task description. Teresa XML is composed of various languages: it embraces an abstract user interface description language that is refined with platform-dependent aspects by a concrete user interface description.

A limiting factor in Teresa XML are its hard-coded elements for particular presentations: the DTD defines `concrete_desktop_interface` and `concrete_mobile_interface` for example. Although this limits the expressive power of the language, it does support a more sensible approach in designing the user interface for other platforms. A typical example is the concrete vocal interface that specifies a speech-driven interface for the task specification. The Teresa tool allows to specify for each task the platforms that can support the task.

Teresa XML is focused on the *presentation model*, but integrates smoothly with the ConcurTaskTrees XML-based *task model* language. The ConcurTaskTrees XML is also considered a part of Teresa XML. Teresa XML also provides two levels of abstraction of the presentation model: abstract and concrete user interfaces. The *dialog model* is specified by a set of transition rules that are also expressed in XML. In ConcurTaskTrees XML there is support for the domain model, but it is unclear how this integrates with Teresa XML or how the user interface description can connect to application objects.

To summarize: Teresa XML integrates several models. The task and presentation models are explicitly supported, the dialog model is closely related to the task model. It is possible to specify the domain objects: the Teresa

tool allows to relate a set of objects with a task. These objects should be described in terms of class (data type), interaction type, access mode and cardinality. Teresa XML implements two of the four presentation model aspects: the structure of the user interface is described (an "interactor composition" container is used for this purpose), widget mappings are done based on the output platform (desktop, Personal Digital Assistant (PDA), mobile phone and voice). There is no support for rendering hints in the current release of the tool, but some preliminary work to support rendering hints is published in [CMP04].

3.3.5 Interface Specification Meta-Language (ISML)

The Interface Specification Meta-Language is created by Crowle and Hole [CH03, Cro04] as an XML-based, metaphor oriented, User Interface Description Language for the design of graphical interfaces (`http://decweb.bournemouth.ac.uk/staff/scrowle/ISML`). The framework is built around five concepts: devices, components, interactors, tasks and meta-objects. A certain degree of device-independence is obtained by the abstraction the designer can provide for an input/output device. Metaphors are the basis of a ISML design artifact; the ISML website states:

> *ISML has been designed on the basis that the metaphor is an independent and partial mapping between a model of tasks understood by the user and the computational operations on the application domain by the underlying system. Arguably, the metaphor mechanism that acts as a bridge between the system and the user's world of work has only partial correspondences with each domain.*

In its current form, ISML is a fairly complex language that requires some background in HCI design since it relies on specific notions found in the classic HCI literature. E.g. the language is *metaphor* based, so the designer should have a good understanding of the level of abstraction a metaphor can offer in HCI design. One of the main benefits of the ISML framework is the particular abstractions it provides: it is one of the only specification languages that integrates abstractions for both input and output devices. This could be useful for designing an ubiquitous interactive system for example.

A binding with application logic is accomplished by the component concept: it allows to describe external application logic that is available, what kind of data is going in and out the application (provided by the input devices) and a state machine to model the behavior of the interface. Metaphors

are described by meta-objects and mapped on a concrete representation by the interactors. This mapping can be interpreted as a AIO to CIO mapping. Finally tasks are specified by re-using the meta-objects, resulting in a hierarchy of linked nodes that represents the sequence of actions executed using the metaphors defined in the meta-objects. As a side effect, the mapping between metaphors and interactors results in an interface that supports the specified tasks.

To summarize: ISML is a complex and sophisticated User Interface Description Language that has a very steep learning curve. There are a lot of different aspects being modeled in the language so ISML would certainly benefit from a simplification or tool support. On the other hand: the framework goes beyond widget sets and uses metaphors and device abstractions to create a user interface. This makes it more suitable for less traditional user interfaces such as the one typically found in ubiquitous systems. ISML implements three of four aspects of the traditional presentation model. The structure of the interface is determined by composing interactors that actualize the meta-objects. The widget mapping is just one of the mappings that exist between meta-object, interactors and component devices. Also ISML components can be mapped onto concrete visualizations. Layout is not supported separately, but as part of the set of meta-object constraints that can be specified. Rendering hints that can define a style for the final, concrete interactors are not supported (although they could be specified as constraints). It seems the interactors and meta-objects are the concepts from ISML that cover most of a traditional presentation model.

3.3.6 The User Interface Markup Language (UIML)

The User Interface Markup Language (UIML) by Abrams et. al. [AH04b, Pha00, FPQAS02, APQA04, AH04a] is perhaps the most well-known and widely spread general XML-based User Interface Description Language. A more extensive discussion of UIML can be found in chapter 9, where we present our own implementation of UIML version 3.1 [AH04b]. We will limit ourselves to some highlights of UIML that make it possible to compare it with the other High-Level User Interface Description Languages presented in this chapter.

UIML can be used as a notation for the *presentation model* and *partially the domain or application model*, but has no explicit support for a task or dialog model. Bleul et. al, introduced the Dialog and Interface Specification Language (DISL) that extends UIML with a dialog model [BMS04]. This approach uses the Dialog Specification Notation to specify the different dialogs

that can appear as part of the user interface. It is based on the concept of a state machine: transitions between states (=dialogs) are made explicit as a part of the behavior section of a UIML fragment. In section 9.7 we provide our own solution how UIML could be extended with a task and dialog model without changing the language itself.

An important advantage of UIML over the other High-Level User Interface Description Languages is its expressive power: UIML is an XML-based *meta*-language. It does not specify any particular widgets or widget sets, but allows to define custom vocabularies with mappings from the desired abstractions and properties onto concrete widgets and widget properties. Using the same vocabularies the application model can be specified by adding an XML description of the interface it offers ("logic").

To summarize: UIML implements all four aspects of the presentation model: it defines the structure (parts) of the user interface and the rendering hints (style and properties) in separate sub-parts of the same interface description. Widget mappings are specified separately in a vocabulary ("peers"). The layout can be described using the properties and structure; there is no separate concept for the layout description however.

3.3.7 XIML

XIML [PE02] is an XML-based specification language for interactive systems [PE04]. It goes beyond most of the discussed languages by providing support for most aspects of a software engineering cycle. XIML supports five different models: task, domain, user, dialog and presentation. For each supported model, a separate subtree in the XML document is created (each model has its own tag-name). Extensive use is made of generic relations between elements to integrate the different models with each other.

XIML targets multi-device user interface design. For this goal, it provides a strict separation between the user interface definition and rendering of the user interface. This is a common approach in most Model-Based User Interface Development, although this separation was not always clearly defined.

Unfortunately the XIML specification is not free: it is only accessible if one agrees with a research-license agreement. For this reason we never had the chance to study the XIML schema's in detail. XIML has been a widely recognized initiative, and many research papers that introduce some new XML-based User Interface Description Language refer to XIML. It was also to first one that supported a complete Model-Based User Interface Development environment.

To summarize, XIML is a complete XML-based User Interface Description Language and could also be a universal language if it offers custom abstraction instead of abstractions of a set of predefined interactors. Since there are too few examples available, we have no information about the aspects (structure, layout, rendering hints and widget mappings) XIML supports. Although XIML is more than just a presentation model, the conclusions only apply to this one model.

3.3.8 UsiXML

UsiXML by Limbourg and Vanderdonckt [LVM+04a, LVM+04b] is an XML-compliant User Interface Description Language that describes the user interface for multiple contexts of use (`http://www.usixml.org`). In contrast with most other approaches mentioned in this chapter it encapsulates different models and has explicit support for relating different models, transforming models and selecting parts of a model. Graph transformations are the central mechanism that can be used to manipulate the diversity of models that can be expressed in UsiXML. Each model that is defined in the UsiXML language can be used separately from the other models; this gives control to the designer about what she/he will or will not include in the description of the interactive system. Multi-path development of the user interface is emphasized: it tries to cover all possible ways in which a user interface can be developed using different models.

For now, UsiXML is the most well-developed and expressive XML-based User Interface Description Language because it supports multiple models (which can be used separately), forward and reverse engineering, and it is built on a firm formal foundation. It is limited in the specification of its presentation model however: in contrast with UIML there is no mean to introduce a custom abstraction (and mapping for this abstraction) for the abstract interaction objects.

UsiXML defines the following models: a domain model, a task model, an abstract user interface model, a concrete user interface model, a context model, a mapping model and a transformation model. The transformation model defines a system to transform one model into another one that has a different type, or in the same type of model.

To summarize: UsiXML is a complete user interface design language that has a visual representation and a formal foundation. If we compare UsiXML to the other User Interface Description Languages discussed in this chapter, the comparison is based on the presentation model. Although UsiXML is

more than just a presentation model, the conclusions only apply to this one model which can be mapped onto the abstract user interface model, concrete user interface model and part of the mapping model in UsiXML. For the presentation model UsiXML supports all four aspects: widget mappings are clearly available (since there is a powerful transformation system in UsixML), structure is supported by a recursive container that can be used to group AIOs, rendering hints are limited to the attributes that are defined for each interactor in the concrete user interface model and layout is supported by alignment specifiers for graphical user interfaces.

3.4 Discussion

Now that we have introduced several different XML-based User Interface Description Languages, we can analyze how they relate to each other on several aspects. The aspects *expressive power*, *number of widget set-related tags* and *user interface coverage* have a well-defined relation, like we will show in figure 3.1. As long as there is a need to pre-define a set of widget set-related tags, the User Interface Description Language is no universal User Interface Description Language. It is constrained by this predefined set. We expect the expressiveness of the User Interface Description Language to grow with the size of the predefined set of widget set-related tags. This is true but has a negative influence on the complexity of the language. When a User Interface Description Language has *no* widget set-related tags it has a maximum coverage because it is not bound by any predefined interactor. Figure 3.1 situates the different User Interface Description Languages in a two-dimensional space that compares their interactor coverage versus the number of predefined interactor tags. UIML and ISML are examples of User Interface Description Languages that have a maximal coverage. Tables 3.1 and 3.2 compare the different languages w.r.t. their properties and integration of different models.

In spite of the time that has passed since the conception of XML-based High-Level User Interface Description Languages (see section 3.2), there is no convergence towards a standard notation (defining both syntax and semantics), the many different initiatives suffered from scattering and no means for exchanging experiences and information. Some open questions can be identified that should be tackled to advance towards more mature XML-based High-Level User Interface Description Languages. These open questions could be divided into three clusters:

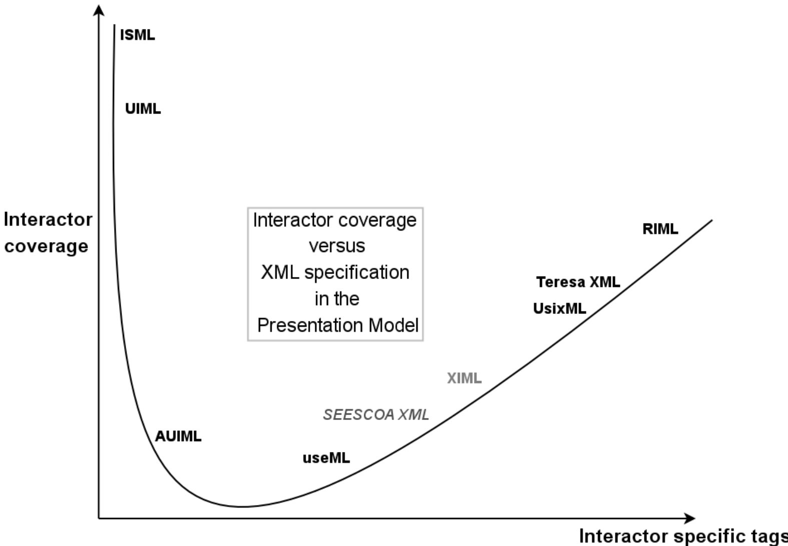


Figure 3.1: Interactor coverage versus number of interactor specific tags

1. **Abstraction and meta-language**: Is there a formal (unified) model possible for XML-based User Interface Description Languages? Can we define a gradation in the abstraction the language offers? How does the language support separation of concern? Given a software engineering process, can the XML-based User Interface Description Language be integrated in this process and even be a standard part of this process?

2. **Evaluation and Coverage**: What are the best ways to ensure a usable user interface that results from a user interface specification in an XML-based User Interface Description Language? For which range of applications is the XML-based User Interface Description Language usable/suitable? What kind of functionality is provided by the supporting tools? What language syntax and semantics are "better" and why? How complex (/readable/usable) is the XML-based User Interface Description Language?

3. **Standardization**: is there one unified language? Which parts of a generic XML-based User Interface Description Language are candidates for standardization? What is common in the existing XML-based User Interface Description Language and how to choose the best for standard-

ization? Should there be a relation between the types of applications that can be supported ("vertical" facilities) and the XML-based User Interface Description Language ("horizontal" facility)

Notice these clusters can be used to evaluate an XML-based High-Level User Interface Description Language. Most XML-based languages have a firm focus on reusability and scalability: making one design for multiple devices is the main goal. This is something that has been quite successful for form-based interfaces, but more multi-modal or even graphically challenging interfaces that are usable are not supported very well yet.

Table 3.1: Overview of selected properties of XML-based High-Level User Interface Description Languages

Language	*Abstraction*	*Coverage*	*Standardization*	*Availability*
	⋆: concrete widgets, ⋆⋆: predefined abstractions, ⋆⋆⋆: user-defined abstractions	⋆: Presentation or task model only, ⋆⋆: ≤ 2 models, ⋆⋆⋆: ≥ 3 models	⋆: No standardization, ⋆⋆: Preliminary standardization efforts (e.g. Oasis), ⋆⋆⋆: Official standard (W3C, IEEE,...)	⋆: only in publications, ⋆⋆: schema free available, ⋆⋆⋆: schema and tools free available
AUIML	⋆⋆	⋆⋆	⋆	⋆⋆⋆
RIML	⋆	⋆	⋆⋆⋆[1]	⋆⋆⋆
useML	⋆⋆	⋆	⋆	⋆⋆⋆
Teresa XML	⋆⋆	⋆⋆⋆	⋆	⋆⋆⋆
ISML	⋆⋆⋆	⋆⋆	⋆	⋆⋆
UIML	⋆⋆⋆	⋆	⋆⋆	⋆⋆⋆
XIML	⋆⋆⋆[2]	⋆⋆⋆	⋆	⋆
Usixml	⋆⋆	⋆⋆⋆	⋆	⋆⋆⋆
SEESCOA XML	⋆⋆	⋆	⋆	⋆

[1] RIML uses existing W3C standards
[2] Assumption: could not be verified

Language	*Task Model*	*Dialog Model*	*Domain/Application Model*	*Presentation Model*
AUIML			⋆⋆ (tag binding with Java classes)	⋆⋆ (Predefined set of interactors)
RIML			⋆⋆ (XForms type data-binding)	⋆⋆ (Predefined set of interactors)
useML	⋆ (Five fine-grained interaction types)			⋆⋆ (Predefined set of interactors)
ISML	⋆⋆	⋆⋆ (state transition network)	⋆⋆	⋆ ⋆ ⋆ (metaphor based)
UIML			⋆⋆ (Object Oriented)	⋆ ⋆ ⋆ (Generic mappings)
XIML	⋆⋆	⋆⋆	⋆⋆	⋆⋆ (Generic mappings)
Teresa XML	⋆⋆ (ConcurTaskTree XML)	⋆⋆ (Transition Rules)	⋆⋆ (ConcurTaskTree XML)	⋆⋆ (Supports a set of interactors in multiple modalities)
Usixml	⋆ ⋆ ⋆ (Generic Task Model)	⋆⋆	⋆⋆ (Object Oriented)	⋆⋆ (Predefined set of interactors)
SEESCOA XML			⋆ (action protocol)	⋆⋆ (Predefined set of interactors)

Table 3.2: A model-based comparison of XML-based High-Level User Interface Description Languages. ⋆: model has limited support, ⋆⋆: predefined model is supported, ⋆ ⋆ ⋆: generic model is supported. No ⋆ indicates the model is not supported.

Part II

HCI Engineering, Models and Transformations

Chapter 4

Dygimes: Dynamically Generating Interfaces for Mobile and Embedded Systems

Contents

4.1 Introduction 51
4.2 Dygimes process 53
4.3 XML-based User Interface Descriptions 54
4.4 Task Model 56
4.5 The System Glue: an Interaction and Application Model 60
4.6 Automatic Layout Management 63
4.7 Customization and Templating 63
4.8 Towards a Tool Chain to support Model-Based User Interface Development 65
4.9 Discussion 68

4.1 Introduction

As a part of the research for this dissertation a framework for dynamically generating User Interfaces for embedded systems and mobile computing devices has been developed, which is presented in this chapter. The main purpose of the framework is to ease the work of the mobile and embedded user interface designer and implementor, and to provide support towards the design of ambient and pervasive interactive systems. The term framework is coined here

as a library with a set of supporting tools on top of which user interfaces can be built and deployed. It is not a code framework from the designers point of view, since the designer can use the Dygimes framework without writing any program code. Often the software implementor of an embedded system or mobile computing device also takes care of the design and implementation of the user interface for the system. This is mainly due to the device specific constraints that have to be taken into account: a thorough knowledge of the device is necessary. The Dygimes framework is conceived to ease the creation of the user interface without the need for specific knowledge of the hardware or software platform. Runtime transformations of user interfaces for adaptation to the target device are also supported. The Dygimes framework supports dynamic generation of user interfaces in this sense it can automatically (at run-time) compose user interfaces from information it retrieves from the specifications and the host platform. E.g. we will show it can compose a user interface dynamically from an annotated task specification. We will also show it has an automatic layout management system that can change the presentation of the user interface according to the target device without any manual intervention.

Besides an overview of the framework and process, this chapter also provides an overview of the models we use, and how they are integrated into one Model-Based User Interface Development environment. The following chapters will elaborate more on the details of the different models, the relations between these models, the algorithms that allow the generation of a (prototypical) user interface and the use of Dygimes in industrial settings. In part III we will show the newest and future developments based on the Dygimes framework, like context-sensitive and distributed user interfaces.

To summarize what will be discussed in the next sections: Dygimes is a framework for generating multi-device user interfaces at runtime. High-level XML-based user interface descriptions are used, in combination with a task specification, an interaction specification and spatial layout constraints. The high-level XML-based User Interface Description Language contains the Abstract Interaction Objects that are included in the user interface. These Abstract Interaction Objects are mapped to Concrete Interaction Objects[VB93]; the mapping process can be guided by providing customized mapping rules.

The user interface creation process is introduced in section 4.2, and discusses the required components to build multi-device user interfaces for embedded systems and mobile computing devices. These components are discussed into detail in the following sections. First, XML-based user interface descriptions are introduced in section 4.3 as the basic building blocks for the

process. Continuing with section 4.4, the use of a task specification will be explained, and the relation between the task specification and the XML-based user interface descriptions. Section 4.5 shows how the created user interface can be attached (or "glued") to the interfaced functionality it presents in a location-independent manner. Section 4.6 shows how the system can ensure consistent user interfaces. This chapter also concludes with a discussion in section 4.9.

4.2 Dygimes process

As stated in section 4.1, the main purpose of the framework is to ease the work of the user interface designer *as well as* the work of the application implementor. At the same time a clear separation between the work of the designer and the work of the implementor is supported. This is desirable because of the pitfalls involved in implementing user interfaces for embedded systems and mobile computing devices. These user interfaces require specific knowledge about the device and the software platform available for that device.

Throughout this chapter we will use a case study, managing a simple publication database, to show how the framework helps the user interface designers and system implementors. The database requires the user to login before using the system. The system offers roughly two different kinds of tasks: adding a paper to the database or searching for a paper in the database. Both require some information to complete the tasks successfully. We kept the example intentionally simple for illustration purposes.

A task specification for this task is developed, enriched with the user interface building blocks (see chapter 8). This will be sufficient to generate prototype user interfaces useful in a user-centered design process. The time needed to create these prototypes is extremely short because many of the steps the designer had to do manually with traditional GUI building toolkits are now automated by the framework. For example the transformation from the task specification to the resulting functional user interfaces built by the user interface designer is done automatically, based upon the algorithm discussed in chapter 5. A *micro-runtime environment* offers support for rendering the created user interfaces independent of the chosen widget toolkit. Section 4.4 will show how all user interfaces stay consistent with regard to the task specification. A graphical overview of the user interface construction and rendering process is shown in figure 4.1, all parts (including how the actual communication with the functionality takes place) will be explained in the following sections. Everything inside the dotted lines is done automatically: the

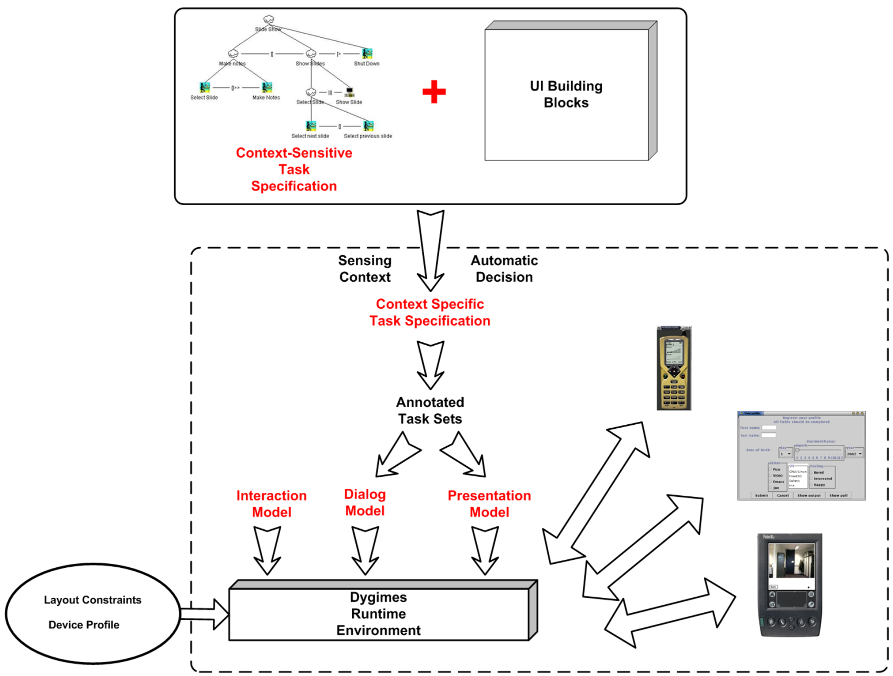


Figure 4.1: The Dygimes process for creating mobile and multi-device User Interfaces. Everything inside the dotted lines is done automatically.

designer inputs exists out of the task specification, the user interface building blocks, the layout constraints. It is also the designers responsibility to relate the user interface building blocks with tasks from the task specification. Notice context detection is not included in the runtime environment itself, but still part of the non-interactive part (chapter 10 will explain how context is being used with the Dygimes environment).

4.3 XML-based User Interface Descriptions

In accordance with the recent growth in mobile computing devices usage, the demand for more suitable multi-device user interface building toolkits also increases. The reuse of existing user interface designs for new devices is problematic: new devices have other constraints making the reuse of these designs very difficult. In contrast consistent look-and-feel is very important, as it contributes to creating a “brand” for the products and makes it easier

for customers to use the new device. To enable flexible reusability of existing designs, we need to abstract the way the user interface is created for a device in a way it becomes less dependent on device-specific properties.

One way of doing this is the use of high-level XML-based user interface descriptions. There are already several propositions and real world examples of the usage of XML to describe user interfaces for multiple devices, most of them are described in chapter 3. To give the reader an idea of which kind of XML-based user interface descriptions are used in our system, listing 4.1 shows the specification of a simple login-dialog. A more profound discussion of this XML language is given in chapter 6. Section 4.5 discusses how generated events are handled and section 4.6 discusses the spatial layout constraints. When the renderer (the runtime environment) processes the description shown in listing 4.1, it can produce concrete user interfaces for different target platforms (shown in figure 4.2) *without* any human intervention.

Listing 4.1: The login dialog user interface description

```
<ui>
  <group name="login">
   <group name="userinfo">
     <interactor>
      <textfield name="login">
        <info>login</info><text size="10"/>
     </textfield>
     </interactor>
     <interactor>
      <textfield name="passwd">
        <info>password</info><text size="10"/>
      </textfield>
     </interactor>
    </group>
    <group name="control">
     <interactor>
      <button name="in"><info>Log In</info></button>
     </interactor>
     <interactor>
      <button name="reset"><info>Reset</info></button>
    </interactor>
   </group>
  </group>
```

```
</ui>
```

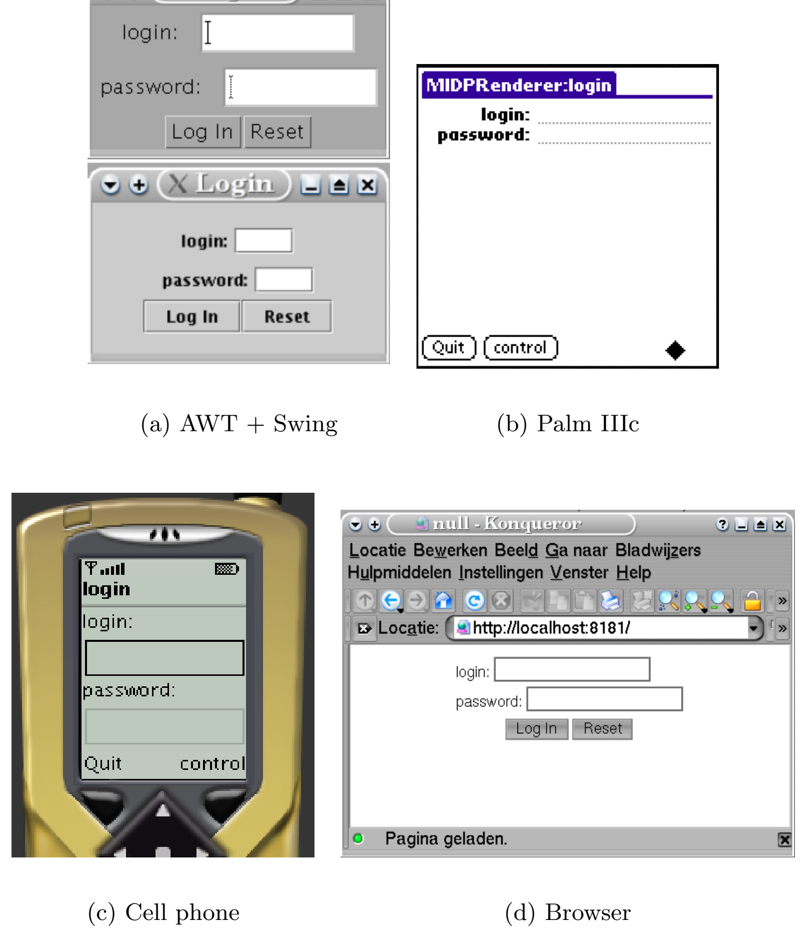
Notice the XML description allows to hierarchically group AIOs using the **group** tag. This way all groups of AIOs that logically belong together are put in the same physical space (e.g. in the same panel or window). At the lowest level, all AIOs in a group should always be presented to the user together. The hierarchical structure of the user interface description allows to recursively group parts of the user interface, i.e. groups can contain other groups, which in their turn can contain other groups themselves. In chapter 6 we will show how the hierarchical structure of the user interface specification with XML and the explicit notion of "groups" can be used to generate concrete presentations for different (and multiple) devices.

This kind of adaptability is important to support device independent authoring. An advanced algorithm to support device independent authoring is provided by the MUSA system by Menkhaus and Fischmeister [MF04]. They show how a richer semantics of the dialog model (the event handler graph) can contribute to more suitable multi-device user interfaces. In the next sections we show how we combine this presentation model with a task model and dialog model to obtain suitable multi-device user interfaces.

4.4 Task Model

The design of a consistent interface starts at the task level. There are several advantages of using a task specification: better requirements capturing, consistent and detailed interface design and better integration with real-life situations [DFAB04]. Nevertheless, software developers seldom use task specifications to develop user interfaces for embedded systems and mobile computing devices. One of the main reasons is the *wide gap* between the implementation of the user interface with its specific device-dependent constraints, and the task specification. To make task modeling more attractive there should be a glue to overcome the gap between the technical challenge of realizing the concrete user interface and designing it with help from a task model. The framework presented here will offer such functionality.

Chapter 2 already discussed task modeling in general and the ConcurTaskTrees notation in specific. The framework discussed in this dissertation uses the ConcurTaskTrees task model proposed by Fabio Paternò [Pat00]. This notation offers a graphical syntax, an hierarchical structure and a notation to specify the temporal relation between activities. For illustration purposes, a simple ConcurTaskTrees specification of the paper-database example is shown



(a) AWT + Swing (b) Palm IIIc

(c) Cell phone (d) Browser

Figure 4.2: The login dialog, rendered for several devices

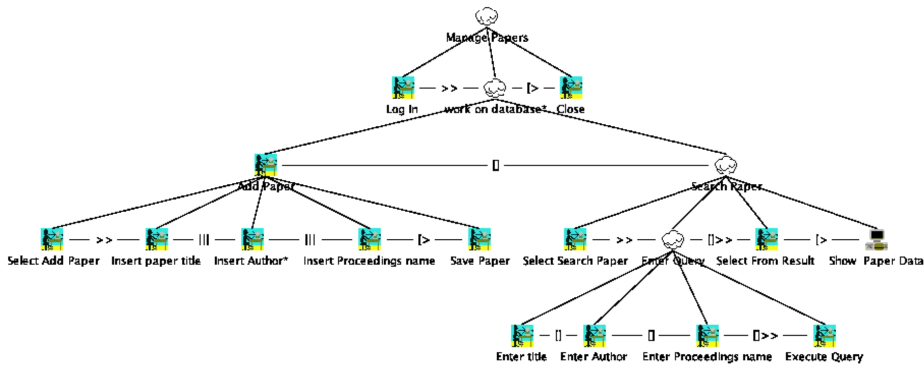


Figure 4.3: Managing a simple publication database

in figure 4.3. A lot of current research extends the ConcurTaskTrees notation for multi-device task specifications. More details about this notation and how we use it are given in chapters 1, 2 and 5.

To convert the task specification into a concrete user interface, the enabled task sets have to be calculated (see section 5.5.3). The enabled task set generation is used as a glue between the realization of the user interface and the task specification. This approach has also been described in [MPS03, PS02] where the focus is on the design of user interfaces, whereas we concentrate on runtime support for creating the user interfaces dynamically using widget toolkits as well as markup languages for the resulting user interface.

The task models can be constructed with the ConcurTaskTrees tool[Pat00], but we use our own enabled task set calculation algorithm: in our approach the ConcurTaskTrees model is annotated with extra information. The algorithm for calculating the enabled task sets from a task specification is included in the Dygimes framework; the different task sets are generated by the runtime environment. The designer does not have to check whether every possible user interface is created for covering each aspect modeled in the task specification: this is done automatically at runtime by the framework through the use of the enabled task sets.

The paper database example illustrates this: its ConcurTaskTrees specification is saved as an XML file and loaded in the annotation tool where the specification can be decorated with XML-based user interface descriptions. Figure 4.4 shows a proof-of-concept tool we developed, it shows the appropriate building block linked to the "Log In" task. On the left-hand side there

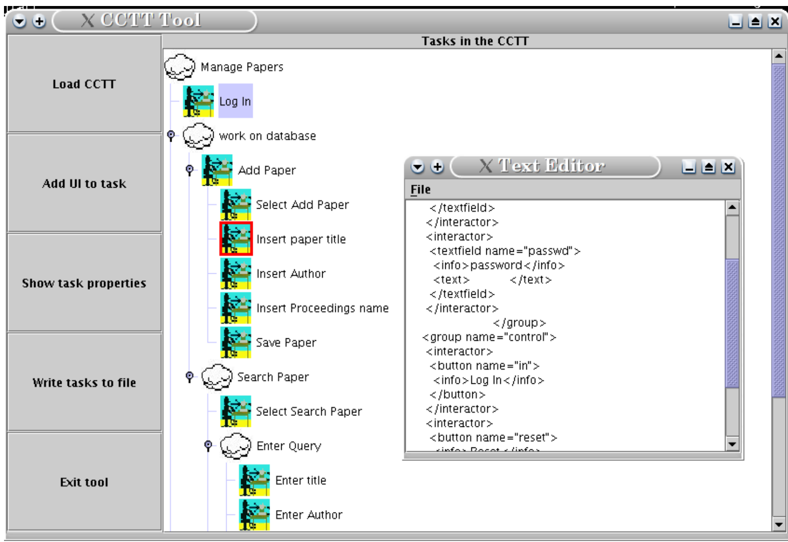


Figure 4.4: The ConcurTaskTrees annotation tool

is an overview of the task specification. The designer can select a task (a leaf) and select the user interface building block (on the right-hand side) that can present this task. This can be selected from a repository of pre-existing building blocks, or loaded from an external XML document by the tool. This supports better reuse of user interface specifications, since the relation between the task leafs and the user interface building blocks can be specified by the designer and the user interface building blocks can be easily used again for another task leaf. The tool stores all user interface building blocks that were loaded in a central repository on the hard disk. When the designer wants to annotate another task specification, formerly created building blocks are also available. This tool is not an interactive graphical design tool to create the user interface building blocks, but only relates the building blocks with the task specification. To create these building blocks the tool shown in figure 4.7 can be used.

4.5 The System Glue: an Interaction and Application Model

Once the designer is satisfied with the user interface, the next step is to "attach the user interface to the application". More particularly we need to provide a mechanism in which the user can interact with application logic through the generated user interface. The application logic is described in an application model. Most Model-Based User Interface Development approaches are attached to a particular application model, like an entity-relationship model or an object-oriented model to describe the application objects. Our approach aims to be independent of the application model as well as to be independent of the location of the application objects. These two requirements are important for multi-device User Interface Description Languages, e.g. a user interface should be able to migrate to another device without the application objects providing the functionality has to migrate together with the interface. Thus an important property for the provided interaction mechanisms is the support for *location transparency*: when a mobile device is used, the implementation of the functionality does not have to be on the same device as the user interface presenting this functionality. Section 4.2 also emphasized that we intend to enable the separation of user interface design and system implementation of embedded systems and mobile computing devices. This means we also need to support several ways to exchange interactive messages between the user interface and the application logic, which could be local or remote.

To overcome these problems, the framework offers an extensible "action-handling" mechanism [VLC03b]. This mechanism is the glue between the user interface and the functionality that will be invoked by the user interface. Because a clear separation between the user interface and the application logic is needed, Dygimes only needs to know which functionality can be invoked on behalf of the logic and which interactors can be used to execute interactions. It does not need to know how the logic implements the functionality (code encapsulation) or where the implementation resides (location transparency). Even the used technique to invoke the functionality needs to be adaptable. To accomplish these goals, we use interaction descriptions that represent the functionality offered by the application.

In Dygimes, an interaction description can be based on the Web Services Description Language (WSDL, [con01e]). This technology allows the developer of the application logic to describe the operations, messages and data types that are supported by the application using existing WSDL editing tools. However, Dygimes also needs a binding between the interaction de-

scription and the abstract user interface. This binding provides Dygimes with the information needed to determine what must happen when a particular event occurs. For this reason we added a section to the interaction description that describes in what way the generated user interface will be bound to the application logic. Suppose, for example, a user pushes the "in" button from listing 4.1. Listing 4.2 then shows what should happen as a response to this action. In this case the loginProcedure operation is sent to the service. This operation is defined in the "paperDBport" portType of the WSDL document. The <uib:parameter> tags describe the parameters. In this case, their values will be extracted from the "login" and "passwd" interactors. It is clear that this kind of description separates the development of the user interface and the application logic and that it supports location-transparent late binding.

Listing 4.2: The binding between an abstract user interface and the application logic

```
<uib:uibinding name="actionbinding" type="paperDBport">
  <uib:interactorbinding name="in">
     <uib:operationlink name="loginProcedure">
        <uib:parameter name="login"/>
        <uib:parameter name="passwd"/>
     </uib:operationlink>
  </uib:interactorbinding>
</uib:uibinding>
```

Dygimes supports different methods to carry out the specified interactions. First, Direct Method Invocation (DMI) can be used to invoke functionality on the application. DMI has the benefit of being fast. However, the drawback with this technique is that DMI can only be used for local invocation with applications implemented in a programming language supporting a reflection mechanism (such as Java). To overcome this problem and to enable location transparency, we make use of web service messaging protocols. These protocols enable us to deploy Remote Procedure Calls (RPC) in an XML-syntax to invoke application functionality. An example of such a technology is the Simple Object Access Protocol (SOAP, [con00]). This protocol uses an XML-syntax to describe which method needs to be invoked upon a web service, together with the method's actual parameters. Those parameters are marshaled from language constructs to XML by using particular serializers. Dygimes also supports XML-RPC[1], which is a more efficient implementation of XML-based

[1] http://www.xml-rpc.com

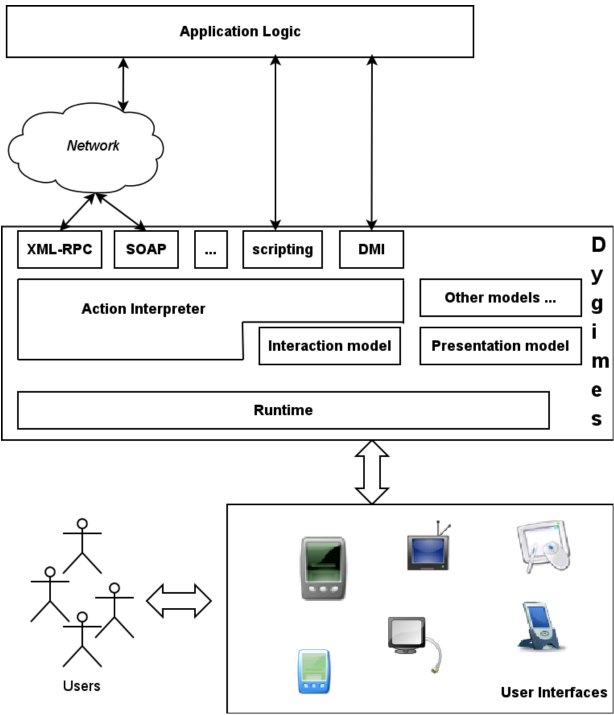


Figure 4.5: The location-transparent action handling glue

RPC. Figure 4.5 shows the extensible architecture of Dygimes enhanced with an interaction model.

A WSDL-based interaction description together with XML-based messaging protocols offer the following benefits:

- Applications become web services-aware through the SOAP implementation. This will be an important advantage in the near future;
- The used approach is device and programming language independent;
- Interaction with remote logic that runs behind company firewalls is supported;
- Common standards for handling interaction are used, namely XML and web services;
- The automatic generation of functional user interfaces for remote applications in a location transparent manner is supported.

4.6 Automatic Layout Management

Abstract user interface descriptions and a constraint-based layout management system are combined in our Dygimes framework for developing adaptive user interfaces for a wide range of devices. When developing a user interface description language that enables us to render the user interface presentation on several different devices, a more flexible approach for laying out concrete widgets than offeredd by the traditional layout management techniques offer is necessary. We focus on screen size constraints because this is the most stringent device constraint in this matter. For example, when a graphical user interface designed for a desktop system has to be rendered for use on a very small screen space (like on a mobile phone) most techniques fail to present a usable interface. The usage of spatial layout constraints can help the designer in these situations: the consistency of the user interface is enforced, yet the user interface is flexible enough for large differences in available screen size. There exist several other constraint-based layout management systems like the ones presented in [Bor79, MBFB89, SMFBB93], but none really focus on providing a flexible layout for embedded systems and mobile computing devices. The layout manager we implemented in the Dygimes framework *does not guarantee* a "visually pleasing" user interface, but makes sure the user interface is suitable and consistent on different devices. This also means the effects on the usability of the user interface are unpredicatble: an automatic rearrangement of the interface can lead to a less usable user interface. This is the price we pay for adding this degree of flexibility to the system. Adding (platform dependent) placement strategies, like the ones presented in [BHLV94], is planned and this can be used to have ensure a certain degree of usability despite the flexbility that is gained. The layout management algorithm is described in chapter 7.

4.7 Customization and Templating

The previous sections explained that the framework uses High-Level User Interface Description Languages descriptions with constraints to render the user interface. The translation from the AIOs into CIOs can be done fully automatically, however this can give unexpected results. For this reason, the Dygimes framework allows the designer to have more control over the rendering of the user interface by allowing them to specify which CIO is used to render an AIO [VLC03a]. Mapping rules can specify mappings for one AIO in one specific interface or they can specify mappings for a range of AIOs. This way, the

designer can define a template, in the form of a set of mapping rules for a certain platform, that can be refined and adapted for specific user interfaces.

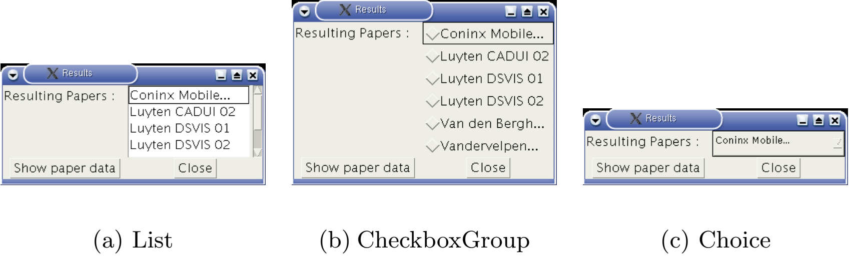
The mapping rules are intentionally kept simple because they are to be used at runtime on devices that can have very limited resources. The CIO that will be used to render a certain AIO depends on the type of the AIO and the name that identifies the AIO. Mapping rules can specify part of the name of an AIO, the complete name of an AIO or no name in order to define their applicability. We will illustrate this with the example of the paper database.

Listing 4.3: Two of the specified mapping rules

```
<mapping>
  <aio2cio>
    <aio>choice</aio>
    <cio>awt.CheckboxGroup</cio>
  </aio2cio>
</mapping>
<mapping>
  <aio2cio>
    <aio>choice</aio>
    <cio>awt.List</cio>
    <name>large</name>
  </aio2cio>
</mapping>
```

For this example, we used a template for the Java AWT platform. Two of the rules in this template are shown in listing 4.3. The first rule shows that the AIO “choice” is mapped onto a CheckboxGroup by default (no name is specified). When the number of items is large or varies over time, as is the case for the AIO that contains the result of the query, a List widget is a better choice. By giving the AIOs a name that contains “large”, the second rule in listing 4.3 is used which gives the wanted result (figure 4.6(a)). If the designer prefers a Java AWT Choice for rendering the query result, this can be indicated by adding a rule that contains the full name of the AIO (figure 4.6(c)).

A feature of the templating system, which is not shown in this example, is that it allows to specify a “null” CIO for AIOs that should not or cannot be represented on a certain platform. This generates a lot of flexibility but can introduce problems as well: when the AIO that cannot be represented is crucial in a certain task, it can render a whole part of the user interface useless. A way to deal with this situation in an effective way, without loosing



(a) List (b) CheckboxGroup (c) Choice

Figure 4.6: Three possible mappings for the query result on the AWT platform

the flexibility of the system, is being worked on. Currently, the designer is still forced to deal with this situation explicitly, by providing an adapted task model for the specific device, as needs to be done in the approach taken by Calvary et al[CCT+02].

4.8 Towards a Tool Chain to support Model-Based User Interface Development

To support the use of Model-Based User Interface Development in the traditional software development process [Som04], we argue small functional tools are more suitable than an integrated environment. Model-Based User Interface Development typically requires the specification of selected models like the task model, the dialog model, the user model and the presentation model. Some of these models are already partially captured in the software process that is instantiated.

Tools to support Model-Based User Interface Development can have a larger impact and a greater success rate if they are built according to the UNIX philosophy. One tool is responsible for exactly one design or development task, and different tools can be used as filters for the output of the other (like UNIX pipes). This should make it easier for tools to integrate in the software process.

Basically, we divide support tools in roughly three categories:

Relationship Management : Tools that can relate different models to each other by different types of relationships. How these relationships are specified depends on the type of models being used.

Transformations : Tools that apply some kind of mapping algorithms on the current model using the information available in the model (e.g. mapping from an abstract to a more concrete model [4]). Transformations include filtering, querying and deducting new relations from the models or even the extraction of a different model based on another one.

Annotations : Tools that allow to add information to the model, that makes it suitable for further processing the model, such as device specific information.

These three categories are similar to the three mechanisms proposed to solve the mapping problem [LVS00]. Transformation tools perform model derivation and composition, while annotation tools can take care of the linking and binding with other models. Notice that for the moment being, our main interest is *not* on providing rich graphical design tools, but on tools for processing selected models that may or may not have a graphical user interface. We believe this is valuable because of the current movement of the software process to support design and development of multi-platform applications: the software developer should be enabled to select the most appropriate models and integrate these in the traditional software engineering process.

We introduce “singular task” tools that are appropriate for one type of task like relating models, transforming models or annotating models with (extra) information. In contrast with most other approaches a complete development environment for the Model-Based User Interface Development approach presented here is not the goal because such an integration can make the separation of concern harder (“who creates and edits which models?”). The tools that were created following the singular task paradigm to support our approach are:

User Interface Building Block Annotator This tool allows to attach XML-based High-Level User Interface Description Languages to tasks of an hierarchical task specification. Figure 4.4 shows the annotation tool that can be used with the Dygimes framework. It can load a ConcurTaskTrees specification and attach SEESCOA XML documents to the tasks.

Runtime Rendering Engine This tool, the “UiBuilder rendering engine”, takes an XML-based High-Level User Interface Description Language and renders it on an output device, taking into account the platform on which it is being used. E.g. it can choose an available widget set and select the most appropriate widget mappings for the current plat-

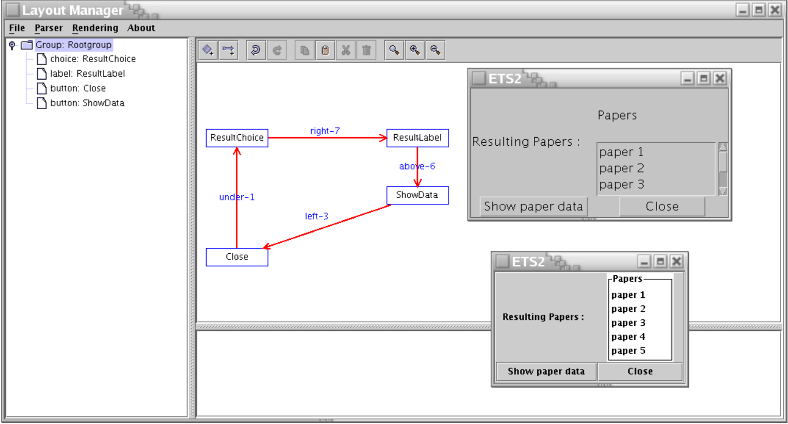


Figure 4.7: Tool for managing spatial constraints. Preview of the user interface is supported.

form. Both Uiml.net (chapter 9) and UiBuilder (chapter 6, the rendering engine for SEESCOA XML) are rendering engines.

Layout Specification Tool Figure 4.7 shows an interactive tool that can load a SEESCOA XML document and specify layout constraints between the different AIOs that are specified [LCC03]. The designer can render the user interface any time she/he wants, since this tools works on the Dygimes framework which includes the UiBuilder renderer. Chapter 7 delves deeper into the way layout can be specified for an abstract user interface.

TaskLib This tool provides a set of transformation algorithms to generate a dialog model from a task model [LCCV03]. The task model should be an hierarchical model and allow temporal relations between tasks. The TaskLib tool is actually a front-end for the TaskLib library included in the Dygimes framework. The TaskLib library implements several algorithms from chapter 5.

In some situations the designer needs an overview of the different models and immediate feedback of her/his manipulations on the models: for this purpose an integrated tool can be built with these different singular task tools.

4.9 Discussion

This chapter served as an introduction and overview of the Dygimes framework. Several aspects were highlighted, such as the process to design and create user interfaces, different models that are used in the process and the vision on (future) tool support for this kind of processes. Dygimes is a framework for creating user interfaces for embedded systems and mobile computing devices. It incorporates several techniques from Model-Based User Interface Development, XML-based User Interface Description Languages, automatic layout management and location transparent event handling. The main purpose is to ease the creation of consistent, reusable and easy migratable user interfaces. These user interfaces can automatically adapt to new devices, offering the same functionality, without being redesigned. If one wants a better adaptation to a particular device, the designer can choose to provide a set of mapping rules and/or a set of better spatial constraints to embellish the presentation of the user interface for that device. Notice the actual user interface does not need to be rebuilt from scratch here.

The next chapters will delve deeper into the different models and features that are touched upon in this chapter.

Chapter 5

Models for Multi-Device User Interfaces

Contents

5.1 Introduction 69
5.2 Related Work 70
5.3 The Task Model within the Dygimes Framework . 72
5.4 ConcurTaskTrees formalism 73
5.5 An algorithm to calculate enabled task sets 77
5.5.1 Introduction 77
5.5.2 Generating a priority tree 78
5.5.3 Calculating the enabled task sets 80
5.6 Activity Chain Extraction 84
5.7 Dynamic Behavior of the User Interface 85
5.7.1 Mapping Sets on States 87
5.7.2 Finding the Initial State 87
5.7.3 Detecting Transitions 87
5.7.4 Mapping the Finishing States 90
5.7.5 The resulting State Transition Network 91
5.8 Actual transitions between dialogs 93
5.9 Discussion 94

5.1 Introduction

The introduction (chapter 1) showed us several models can be used in Model-Based User Interface Development: presentation model, user model, data

model, task model, dialog model,... This chapter will focus on the relation between the abstract models and the concrete models, more specific the task model and the dialog model. We believe this can help to automate the transformation of the specification of the models into a concrete user interface. One of the problems encountered when progressing from one model to another is to keep the different models consistent with each other. When a task specification is used that can specify the temporal relations between (sub)tasks, these relations can be exploited to extract (part of) the dialog model out of the task specification. We believe the dialog model is an essential model to make the transition between the abstract models and the concrete models.

A pragmatic approach will be taken in which feasibility is emphasized over a completely automated transformation from the task specification into a dialog specification. The designer should stay in control of the user interface creation, with the transformation supporting him to keep both models (task and dialog) consistent with each other. The dialog model can be generated by incorporating part of the design knowledge and relying on the modeling concepts available from the task model. We show how the dialog model is governed by an appropriate task model although dialog and presentation model are also closely related. The task design can be of a greater influence on the final presentation of the concrete user interface, making it more suitable for the tasks it should support.

The remainder of this chapter is structured as follows: section 5.2 reports on some significant steps of different dialog models used in methods and tools for user interface development, from the less expressive and executable to the most ones. Section 5.6 explains how an activity chain can be used for extracting a dialog model out of a task model. This is followed by an explanation of the actual algorithm in section 5.5. How the transitions between different windows are invoked, when the dialog model has to be rendered in a real user interface, is explained in section 5.8.

In order to make this chapter more readable, some definitions from chapter 2 will be repeated.

5.2 Related Work

The State Transition Diagram [Par69] was probably the first and the most frequently used formal method to model a dialog, as expanded in State Transition Networks (STN) [Was85]. Other formal methods have also been investigated, but there was no tangible proof of a far superiority of one of them over the other ones with respect to all the criteria defined in [Coc87].

Genius [JWZ93] produced from a data model multiple Dialog Nets, a specific version of a Petri Net for specifying coarse grained dialog in terms of transitions (unspecified or fully specified) between views. The advantage of Petri Nets over previously explored formal methods was that they show the flow with indistinguishable tokens and places and marks can be introduced one at a time.

Tadeus [SE96b] takes envisioned task and object models to infer design of a dialog model expressed in Dialog Graphs, which are both graphically and formally expressed, thus leading to model checking while keeping visual usability.

The Interactive Cooperative Objects (ICO) formalism [BP99], based on Petri nets, allows more expressive and modular dialog specifications than the earlier attempts. In addition, any ICO specification of a dialog can be directly executed, which reduces the gap between specification time and execution time.

Windows Transitions [MV02] also extend STNs so as to create a visual and formal method to specify coarse grained dialog between windows. The STN is based on a series of window operations and transitions and can be drawn on screenshots. However, this model is not generated, but produced by hand. By consequence there is no guarantee to preserve constraints imposed by the task.

Closely related work can also be found in [DFR03, DF04]: the relation between task models and dialog model is closely investigated. Just as in our approach the task model is considered as the starting point of most Model-Based User Interface Development approaches. In contrast with our approach, Dittmar and Forbrig extend and improve the task model so it can represent actions and states. This leads to a better integration between task and dialog models. The differences with our work are the use of different types and views in the dialog graph that is supported in the work of Dittmar and Forbrig which we do not support, and the automatic dialog generation in our approach which is not available in [DFR03].

There is a need to produce a dialog model while maintaining a lightweight approach that supports the designer in creating consistent and correct models. The closest work is probably Teresa [MPS03], which is aimed at producing multiple user interfaces for multiple computing platforms. We also rely on the mechanisms introduced in Teresa and expand them in several ways that will be outlined throughout this chapter. The main differences between the Teresa tool and the Dygimes framework, are that the latter supports runtime creation of user interfaces and the possibility to use different widget set

libraries in addition to mark-up languages. Instead of focusing on tool support the Dygimes framework is focused on automatic creation of user interfaces. While the Teresa tool offers a design environment, the Dygimes framework offers a run-time environment with currently limited tool support. One of the goals is to reuse existing tools for task modeling, so the Dygimes process does not rely on a single monolithic integrated development environment.

5.3 The Task Model within the Dygimes Framework

Within the Dygimes Framework we make extensively use of the task model. It is the first model the designer creates and the Dygimes process defines several steps in which the task model is enhanced and other models are created based on the task model in comprehensible steps. Figure 5.1 gives an overview of the *design* cycle to construct an interactive system. The following steps are included in a design cycle:

1. A ConcurTaskTrees task specification is provided. The leaves in the task specification are annotated with abstract user interface descriptions (user interface building blocks). Graphical tool support to attach the user interface building blocks is provided (see figure 4.4).

2. The task specification and abstract user interface descriptions are merged into one "annotated" task specification. Both the task specification and user interface descriptions can be expressed using XML. This allows a smooth integration and results in a singe XML document the system can process.

3. The enabled task sets are calculated (a custom algorithm to calculate these is provided in the Dygimes framework).

4. The initial Task Set is located (the first Task Set which is shown in a user interface when the application is used, see section 5.7.2).

5. The dialog model can be created using the temporal relations between tasks and the enabled task sets provided in the task model (see section 5.7.3). The dialog model is expressed as a State Transition Network (STN).

6. The Abstract user interface description is generated out of the enabled task sets and the STN. The STN provides the navigation between the

different dialog windows and the enabled task set specifies the necessary content of a dialog window.

7. The transitions are integrated into the user interface description.

8. The actual user interface is generated and shown to the designer or the user.

9. The designer can test the user interface and provide feedback by changing the Compound Task Sets (CTS, see section 5.4) and Abstract Presentation.

10. The Compound Task Sets can be adapted by the designer.

11. Transitions are recalculated according to the new Compound Task Sets.

Although this is not a traditional design cycle as one can find in [DFAB04] for example, it is closely related to transformational development and software prototyping in Software Engineering [Som04].

5.4 ConcurTaskTrees formalism

The task specification central in our approach is the ConcurTaskTrees notation [Pat00], introduced in section 2.3. We tried to take a pragmatic approach: while working bottom-up towards usable algorithms to calculate the Enabled Task Sets and extract a dialog specification from a task specification, we provide some semi-formal definitions.

For a good understanding of the algorithm presented in this chapter, we introduce the following notation. Let $\mathcal{T}$ be an infinite set of tasks. By $\mathcal{R}$ we denote the set of relations { `[]`, `|[]|`, `|=|`, `[>`, `>>`, `[]>>`, `|>`} $\cup\{d\}$ where d is the decomposition relation.

Definition 13 A *ConcurTaskTrees task model* M is a rooted directed tree where every node is a task in $\mathcal{T}$. In addition, there can be arcs between tasks carrying labels from $\mathcal{R}$ modeling connections by temporal operators. An arc labeled o from task t to t' is denoted by $t \xrightarrow{o}_M t'$.

For the remainder of this chapter we fix a task model M. Using the introduced notation we can define a set of tasks of a task model M:

Definition 14 By $\mathcal{T}(M)$ we denote the set of tasks occurring in M.

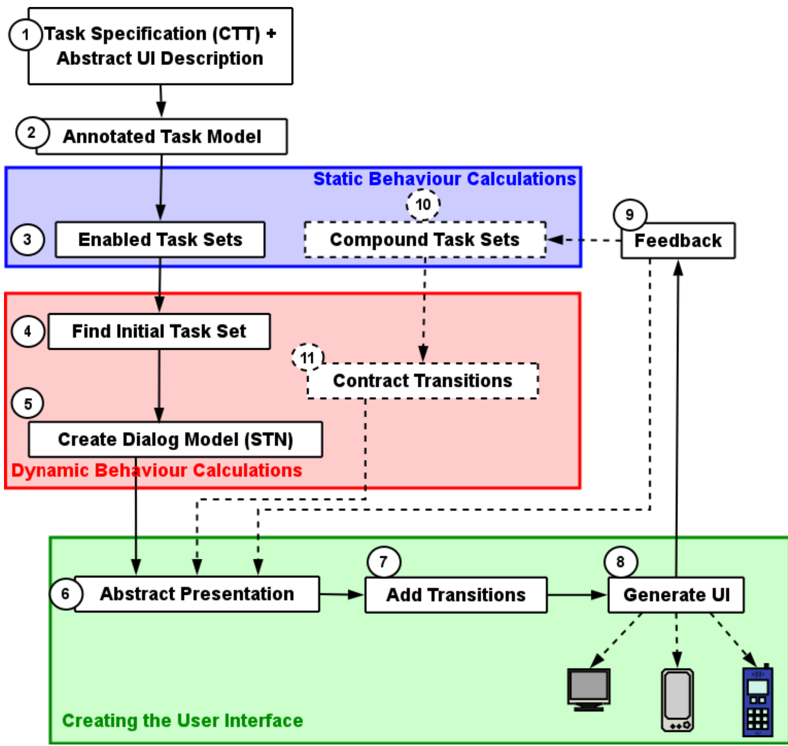


Figure 5.1: The Dygimes User Interface design and generation process

We will define also other relations in this chapter. These definitions have the sole purpose to support the development of the algorithm. A more precise way for defining semantics can be done using Kripke semantics [CGP99] for example.

We will make use of the "mobile phone task specification" for illustration purposes; a task model describing some functionalities offered by a regular mobile phone. It describes the steps to adjust the volume or read an SMS message. The task specification is shown in figure 5.2(a).

A very important advantage of the ConcurTaskTrees formalism is the possibility to generate *Enabled Task Sets* (ETS) out of the specification (see section 2.3). Based on the definition of an enabled task set, an Enabled Task Collection (ETC) E is a set of sets of tasks ($E \subseteq 2^{\mathcal{T}}$). In [Pat00] an algorithm is given to compute a specific enabled task set of a given task model M, we denote the latter by $E(M)$. Usually there are several enabled task sets which

can be calculated out of a task model.

The enabled task sets calculated from the model in figure 5.2(a) are:

$$\begin{array}{l} ETS_1 = \{Select\ Read\ SMS, Select\ Adjust\ Volume, Close, Shut\ Down\} \\ ETS_2 = \{Select\ SMS, Close, Shut\ Down\} \\ ETS_3 = \{Show\ SMS, Close, Shut\ Down\} \\ ETS_4 = \{Select\ Adjust\ Volume, Close, Shut\ Down\} \\ ETS_5 = \{Adjust, Close, Shut\ Down\} \end{array} \tag{5.1}$$

Based on the heuristics given in [PS02], adhesive and cohesive tasks can be defined as follows (definitions 15 and 16):

Definition 15 Two tasks $t, t' \in \mathcal{T}(M)$ are **cohesive** w.r.t. a task model M if there is a set $S \in E(M)$ such that $\{t, t'\} \subseteq S$.

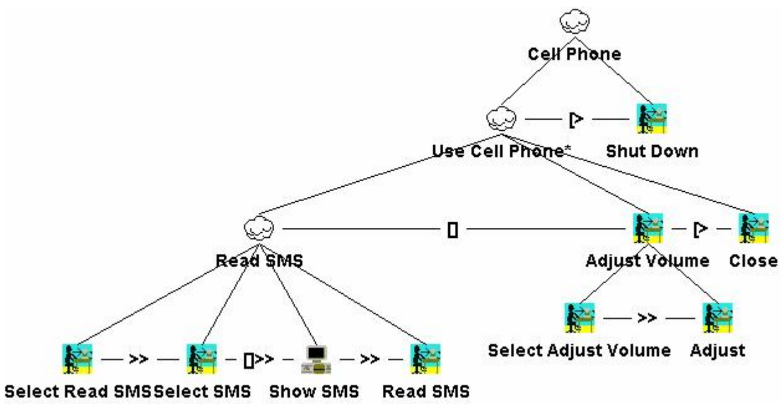
Definition 16 Two tasks, t and t', are called **adhesive** if they are not cohesive, but they are semantically related to some extent.

[PS02] also introduces some rules for merging enabled task sets, which can be very useful when there are a lot of enabled task sets. These heuristics can be used to identify adhesive tasks: two tasks which do not belong to the same enabled task set, but their respective enabled task sets can be merged into one enabled task set. On the other hand, merging enabled task sets can cause problems when a user interface for a mobile phone has to be generated starting from the task model. Typically, a mobile phone can only show one task at the same time due to the screen space constraints. Consequently, a fine-grained set of enabled task sets can ease the automatic generation of user interfaces subject to a (very) small screen space. We defined some heuristics for splitting up enabled task sets in [CLC04d].

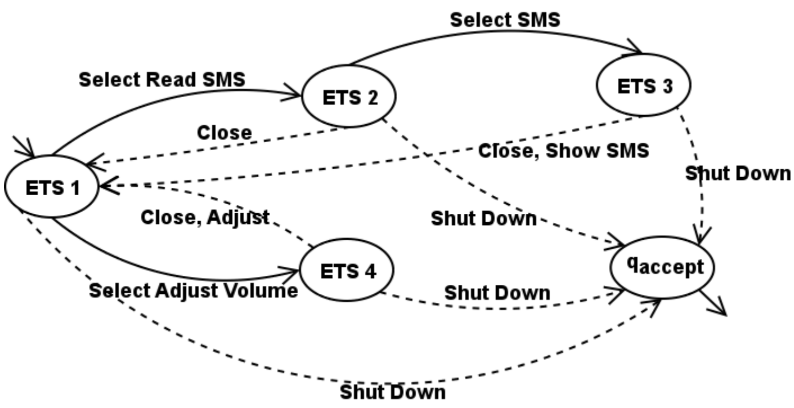
In addition, we define a Compound Task Set (CTS) based on the definition of a task set:

Definition 17 A **Compound Task Set** of a task model M is a collection of tasks $C \subseteq \mathcal{T}(M)$ such that

- every two distinct tasks in C are cohesive or adhesive; and,
- for every $t \in C$ there is an $S_t \in E(M)$ such that $t \in S_t$; in addition, $S_t \subseteq C$.



(a) A ConcurTaskTrees specification for using some functionalities offered by a mobile phone.



(b) State Transition Network describing the behavior of figure 5.2(a)

Figure 5.2: A ConcurTaskTrees diagram for a mobile phone (5.2(a)) and its State Transition Network (5.2(b)).

The different CTSs indicate which user interface building blocks (attached to the individual leaf tasks) can be presented as a group to the user. Notice the composition of a CTS depends on the heuristics the designer applies. In step 9 of figure 5.1, the designer can choose to group certain enabled task sets. Our system relies on the designer to decide what is desirable and what is not. Heuristic rules to contract or split the enabled task sets as proposed in [PS02] or [CLC04d] can be automatically applied by the designer if she/he wishes to do so. Our system provides consistency checks so whenever these heuristic rules are applied, all models that are influenced by these heuristics will update accordingly. The heuristic rules can be used to guide the designer to make better decisions with the next implementation of the Dygimes framework.

The next step is to discover for every $S \in E(M)$ the set $R \subseteq E(M)$ of enabled task sets where every enabled task set in R can replace S when that enabled task set is finished. So, we will try to discover transitions that allow the user to go from one enabled task set to another according to the temporal relations defined in the task specification.

5.5 An algorithm to calculate enabled task sets

5.5.1 Introduction

In our framework, the enabled task sets are calculated by transforming the ConcurTaskTrees specification into a priority tree and applying the predefined rules (somewhat modified) described in [Pat00]. A priority tree is a ConcurTaskTrees specification, where all the temporal relations of the same level in the task hierarchy have the same priority according to their defined order. Such a tree can be obtained by recursively descending into the ConcurTaskTrees specification inserting a new level with abstract tasks where the temporal operators on the same level do not have the same priority. This does not change the semantic meaning. A more formal definition of a priority tree can be found in [CC03] and is repeated here in definition 18.

Definition 18 A *Priority Tree* of a task model M, denoted by $\mathcal{P}(M)$, is a ConcurTaskTree with the same semantic meaning as M but where $\forall t \in \mathcal{T}(\mathcal{P}(M))$ with children $t_1, ..., t_n$ where $n \geq 3$: $\forall i \in \{1, ..., n-2\}$ with $t_i \xrightarrow{o_i}_{\mathcal{P}(M)} t_{i+1}$ and $t_{i+1} \xrightarrow{o_{i+1}}_{\mathcal{P}(M)} t_{i+2}$, o_i and o_{i+1} share the same priority.

Choice	Choice ([])
Parallel Composition	Independent Concurrency (|||), Concurrency with Information Exchange (|[]|)
Interrupt	Disabling ([>)
	Suspend Resume (|>)
Enabling	Sequential Enabling (>>), Sequential Enabling with Information Passing ([]>>)

highest priority ↕ lowest priority

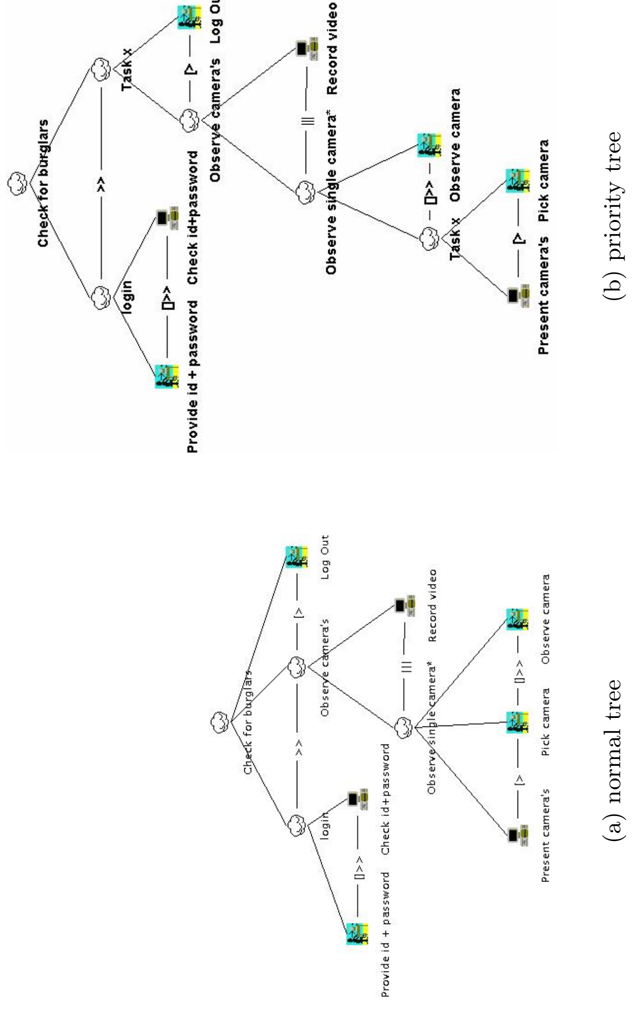
Table 5.1: The priority order of ConcurTaskTrees temporal operators.

5.5.2 Generating a priority tree

For calculating the enabled task sets we used the algorithm described in [Pat00], and made some adjustments to it. Unfortunately, the algorithm given there has some minor bugs, which makes the generated enabled task sets incorrect in some situations[1]. To overcome this problem and to be able to integrate the algorithm in custom tools, our own algorithm for calculating the enabled task sets will be used.

Using the priorities of the temporal operators (sect. 5.4) we can transform the original tree (which represents the task model) into a *priority tree* by using the precedence of the temporal operators. We use the priority table 5.1 to transform a general ConcurTaskTrees into a priority tree (e.g. Interleaving has a higher precedence then disabling). The temporal operators in the ConcurTaskTrees notation have the following precedence: {[]} > {|[]|, |||} > {[>} > {|>} > {>>, []>>}. Using this table we can construct a priority tree: this is a representation of the original tree that has only operators of the same precedence on each level of the tree. The tree is transformed by inserting abstract tasks on the appropriate levels. An example of a regular ConcurTaskTrees tree and its priority tree can be seen in figure 5.3.

[1]Current implementations of the Teresa tool resolve most of the bugs.



(a) normal tree

(b) priority tree

Figure 5.3: A ConcurTaskTrees tree and its priority representation

5.5.3 Calculating the enabled task sets

We need two additional functions for calculating the enabled task sets out of a ConcurTaskTrees specification: *first* and *body*. $first(t)$ identifies the set of subtasks of task t which should be executed first. If t has no subtasks, than $first(t) = t$. $body(t)$ identifies the set of subtasks of task d which are not included in $first(t)$. Notice $first(\{t_1, ..., t_n\}) = \{first(t_1), ..., first(t_n)\}$. The same goes for *body*. In the algorithm a task kan also be labeled with "lfirst" and "lbody", indicating the task should be replaced by a selection of its subtasks according to its label in the next iteration of the algorithm ($lfirst(t)$ is true if t is labeled with "first", otherwise it is false).

The functions *parent* and *children* are also introduced, to make the algorithm easier to read. The *parent* function returns the parent node of a node, if any. For example: $parent(t) = k$ says the parent of the node t is k. The *children* function gives us the children of a node, which is just an expression for the decomposition relation introduced in section 2.3: $children(t) = \{n_1, n_2, .., n_m\}$ where $n_1, ..., n_m$ are the result of the decomposition of t. Of course this implies $parent(n_1) = t$, $parent(n_2) = t$,...,$parent(n_m) = t$. Notice $children(t) = first(t) \cup body(t)$.

Also consider the function $necessaryTasks(S, t)$ with S being an enabled task set and $t \in \mathcal{T}(M)$, this function returns a set with all the tasks of S that have a common disabling parent with t or have certain concurrent relation with t by which we mean directly through temporal operators on the same level or through parents on a higher level in the tree.

Using the priority order from table 5.1, a *priority* tree can be composed from the ConcurTaskTrees model. We obtain the following algorithm to calculate the enabled task sets out of the priority tree:

1. The initial set of enabled task sets $\mathcal{ETS}$ contains a single enabled task set $ets_1 = \{t_r\}$, that only contains the root node t_r.

2. while $\exists d \in ets_i$ where $1 < i < \#\mathcal{ETS}$; and $(children(d) \neq \emptyset) \vee first(d) \vee body(d)$

   (a) If (d is a leaf) do:

      - If($lfirst(d)$): remove the $lfirst$ label from d
      - Else If($lbody(d)$): $\mathcal{ETS} \leftarrow \mathcal{ETS} \setminus ets_i$

   (b) If ($children(d) \neq \emptyset$), take the next level in the tree in which $children(d) = \{n_1, n_2, ..., n_m\}$ participate

      - If ($n_1 \overset{[]}{\rightarrow} n_2$) (choice):

i. $ets_k \leftarrow (ets_i \setminus \{d\}) \cup \{n_1, n_2, ..., n_m\}$ and $\mathcal{E}TS \leftarrow (\mathcal{E}TS \setminus ets_i) \cup ets_k$

ii. $\forall node \in \{n_1, n_2, ..., n_m\}$ make a new ETS and add it to the set of enabled task sets: $\mathcal{ETS} \leftarrow \mathcal{ETS} \cup \{\{lbody(n_1) \cup ets_i\} \cup ... \cup \{lbody(n_m) \cup ets_i\}$

- If $(n_1 \stackrel{|||}{\rightarrow} n_2 \vee n_1 \stackrel{|[]|}{\rightarrow} n_2)$ (parallell composition):
  i. Replace d with $children(d)$ in ets_i: $ets_i \leftarrow (ets_i \setminus \{d\}) \cup \{n_1, n_2, ..., n_m\}$
- If $(n_1 \stackrel{[>}{\rightarrow} n_2 \vee n_1 \stackrel{|>}{\rightarrow} n_2)$ (interrupt):
  i. $ets_k \leftarrow lbody(n_2) \cup (ets_i \setminus \{d\})$
  ii. $ets_i \leftarrow \{n_1\} \cup lfirst(n_2) \cup ets_i \setminus \{d\}$
  iii. $\mathcal{E}TS \leftarrow \mathcal{E}TS \cup ets_k$
- If $(n_1 \stackrel{>>}{\rightarrow} n_2 \vee n_1 \stackrel{[]>>}{\rightarrow} n_2)$ (enabling):
  - If$(lbody(d))$
    i. $\mathcal{E}TS \leftarrow \mathcal{E}TS \cup \{\{n_1 \cup necessaryTasks(ets_i, d)\} \cup ... \cup \{\{n_m \cup necessaryTasks(ets_i, d)\}$
    ii. $\mathcal{E}TS \leftarrow \mathcal{E}TS \setminus ets_i$
  - Else If $(lfirst(d))$
    i. $ets_k \leftarrow lfirst(n_1) \cup (ets_i \setminus \{d\})$
    ii. $\mathcal{E}TS \leftarrow \mathcal{E}TS \cup \{\{lbody(n_1)\} \cup necessaryTasks(ets_i, d)\}$
    iii. $\mathcal{E}TS \leftarrow \mathcal{E}TS \setminus ets_i$
  - Else
    i. $\mathcal{E}TS \leftarrow \mathcal{E}TS \cup \{(ets_i \setminus \{d\}) \cup \{n_1\}\} \cup ... \cup \{(ets_i \setminus \{d\}) \cup \{n_m\}\}$
    ii. $E'(M) \leftarrow E'(M) \setminus \{ets_i\}$

3. Eliminate all User Tasks because they are not shown in a user interface.

When the algorithm described here finishes, we have all the possible enabled task sets from the task specification[2]. Since an enabled task set specifies all tasks that should be "accessible" at one point in time we can map these enabled task sets on dialogs in the dialog model. This corresponds with finding all the possible combinations of presentation units (see definition 9) which should be available in an integrated user interface. Each enabled task set can

[2] In fact there is a post-processing stage involved that removes duplicate entries and removes possible inconsistent tasks from a task set.

be represented by one or more presentation units, so each enabled task set presents one integrated user interface. Following definition 12 we can say this algorithm offers *full coverage* for the task specification.

As an example of the algorithm, consider the ConcurTaskTrees model given in figure 5.3. To calculate the enabled task set out of this model, the model has to be saved into an XML document first. This XML document will be imported into the uibuilder.navigation library (part of the Dygimes Framework), to build an internal representation for this task-model suitable for "UiBuilder", the Dygimes rendering engine. The UiBuilder is a user interface rendering engine which takes abstract XML-based user interface descriptions as input, and renders them in an appropriate form for the target platform [LVC02].

The enabled task sets which are retrieved by using the algorithm described here are: $\mathcal{ETS}$ ={{Provide id + password}, {Check id+password}, {Log Out, Record video, Present camera's, Pick camera}, {Observe camera, Log Out, Record video}, {Pick camera, Log Out, Record video}}. For each element in $\mathcal{ETS}$ an integrated user interface should be provided. Notice this approach makes the transition from a task-model to an effective user interface *easier*, *more consistent* but also *less interactive*.

To make the automatic creation and grouping of the user interface according to the calculated enabled task set work, High-Level User Interface Description Languages are integrated into the ConcurTaskTrees model. This is done by importing the XML document which can be saved with the ConcurTaskTrees Environment in our annotation tool (figure 4.4). The ConcurTaskTrees Environment is a closed source application, so we need to post-process its output, instead of changing the implementation of the environment.

A simple task specification for an email client is presented in figure 5.4; this task specification is created with ConcurTaskTrees Environment. This can be saved as an XML document and imported in our annotation tool. The annotation tool attaches High-Level User Interface Description Languages on the leaf tasks that specify how the task can be presented in a concrete user interface. Saving this annotated specification results in an enhanced ConcurTaskTrees document that also includes High-Level User Interface Description Languages for the tasks it specifies. The result of this step (step 1 from the Dygimes design cycle) is graphically presented in figure 5.5. The enabled

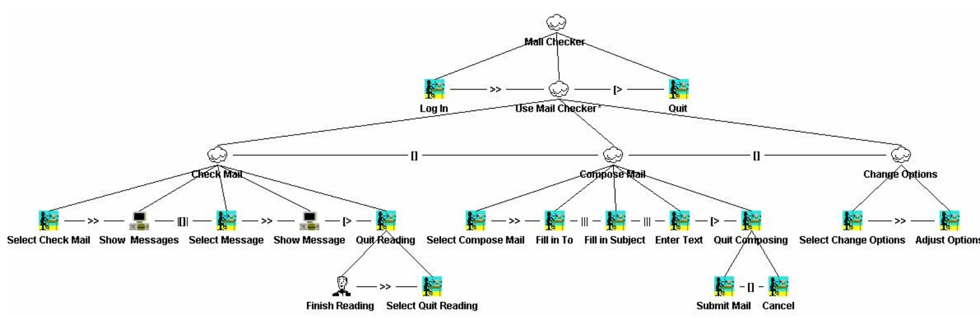


Figure 5.4: A simple email client task specification.

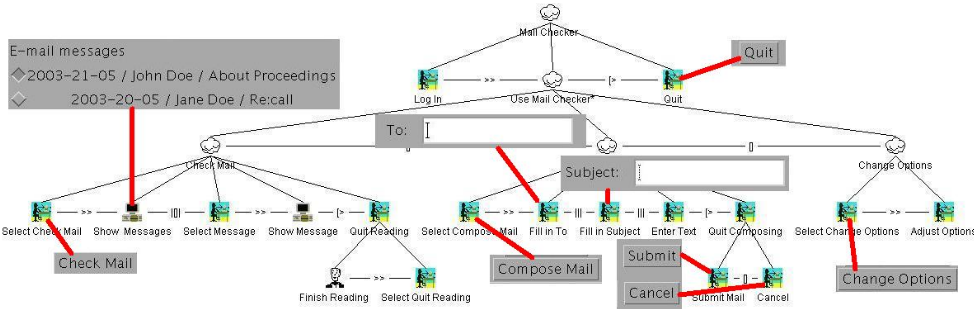


Figure 5.5: A simple email client task specification annotated with user interface building blocks.

task sets from the specification in figure 5.4 are shown in equation 5.2.

$$\begin{aligned}
&ETS_0 = \{LogIn\} \\
&ETS_1 = \{Quit, Select\ Change\ Options, Select\ Check\ Mail, Select\ Compose\ Mail\} \\
&ETS_2 = \{Quit, Show\ Messages, Select\ Message\} \\
&ETS_3 = \{Quit, Show\ Message, Select\ Quit\ Reading\} \\
&ETS_4 = \{Quit, Fill\ in\ To, Fill\ in\ Subject, Enter\ Text, Submit\ Mail, Cancel\} \\
&ETS_5 = \{Adjust\ Options, Quit\}
\end{aligned} \tag{5.2}$$

Full coverage of this set of enabled task sets can be realized by merging the different user interface building blocks attached to each task for each enabled task set. For example, ETS_4 is the aggregation of the building blocks attached to the tasks “Quit”, “Fill in To”, “Fill in Subject”, “Enter Text”, “Submit Mail” and “Cancel”. This results in:

ETS_4= { Quit, Fill in To, Fill in Subject, Enter Text, Submit Mail, Cancel } =

5.6 Activity Chain Extraction

We define an *Activity Chain* as a path of different dialogs to reach a certain goal. A dialog is uniquely defined by the enabled task set which it presents. Each dialog is considered a step in the usage of the application, so a graph of enabled task sets can be built representing the flow of the dialogs the user may see. Each enabled task set is a node in this graph and can have a directed edge to other enabled task sets, which represents the transition of one dialog to another dialog. In addition, a start task set can be identified presenting the initial dialog. Given these properties, the activity chain can be specified as a State Transition Network (STN).

An STN is defined as a *connected graph* with *labeled directed edges*. Edges are labeled with tasks. Nodes are the sets in $E(M)$. In addition, there is an *initial node* and a *finishing node*.

For example, figure 5.2(b) shows the STN for the task model shown in figure 5.2(a). Intuitively, an STN seems insufficient to describe the behavior because of the concurrency supported in the task model. However, tasks which

are executing in the same period of time belong *to the same enabled task set* , which makes concurrent enabled task sets unnecessary. We will only need one active state in the generated dialog specification; STNs are sufficient for this purpose. This condition may not hold when collaborative user interfaces are considered though.

In figure 5.2(b), the transitions between states are labeled with the task names, where *Shut Down* is the exit transition. The goal is to find and resolve these transitions automatically, so the generated user interface is fully active and the system offers support for several related dialogs, without the need to implement the transitions explicitly. By using automatically detected transitions, the user interface designer can make an appropriate dialog model without burdening the developer. It suffices to describe the "transition conditions" when a transition is triggered in the program code. Three different levels of dialogs can be identified: intra-widget (how widgets behave), intra-window (what is the dialog inside a window) and inter-window (how the different windows behave w.r.t each other). The focus in this chapter lies on the inter-window level.

Transitions between states in the STN can be identified by investigating the temporal operators that exist between tasks in the task specification. Usually, one proceeds from one enabled task set to another when an enabling or disabling operator is detected. More formally (based on definition 13), this can be expressed by introducing the following definition:

Definition 19 Let $S_1, S_2 \in E(M)$, t_1 is a **candidate transition** in one of the following cases:

- $t_1 \overset{>>}{\rightarrow}_M t_2$, $t_1 \in S_1$ and $t_2 \in S_2$
- $t_2 \overset{[>}{\rightarrow}_M t_1$, $\{first(t_1), t_2\} \subseteq S_1$ and $body(t_1) \subseteq S_2$

Here, *first(t)* is the first subtask of t that has to be performed, and *body(t)* are the subtasks of t not included in *first(t)*. These two functions are defined in [Pat00] and explained in section 5.5.3.

5.7 Dynamic Behavior of the User Interface

In short, building the STN to guide the activity chain consists of:

1. A set of states; every enabled task set is mapped on a state;

2. A set of transitions; every task involved in a disabling or enabling relation can be a candidate transition as described in definition 19;

3. An initial state; this is the unique initial enabled task set shown to the user;

4. A set of finishing states; the set of enabled task sets that can terminate the application;

5. An accept state; arriving in the accept state will terminate the application. The accept state can be reached out of a finishing state.

The rules we will show here are obtained based on *empirical results*, not by mathematical deduction. This means we can not prove they are correct in every situation, but only know that they work in many situations. As a starting point, our algorithm from section 5.5 is based in the algorithm given in [Pat00]. All the temporal operators are considered allthough there is no generic process: for each priority level (see table 5.1) is treated separately. The algorithm is implemented and tested successfully on several examples, there are only a few cases we know about that give incorrect results. Most of these incorrect results were due to the occurence of concurrency in combination with other operators in the task specification which would lead to an extra ETS. This kind of errors were corrected by adding a post-processing stage in the software which can detect these incorrect results and remove the accordingly.

In the next section, we will show some example rules to extract the STN out of the task specification. This is done in four steps: finding the states of the STN, locating the start state, collecting the transitions between states and finally locating the finishing or "accept" state. The most challenging part is collecting the transitions of the STN: this requires investigating the temporal relations in the task model and deciding which task will invoke another enabled task set.

Before we continue, we define two functions: *firstTasks* and *lastTasks*:

firstTasks : takes a node n of the ConcurTaskTrees task model and returns the left-most group of leaves that are descendants of n and are all elements of the same enabled task set. This function will return a single enabled task set if no ancestor of n is involved in a concurrent relation.

lastTasks : takes a node n of the ConcurTaskTrees task model and returns the right-most group of leaves that are descendants of n. When a concurrency or a choice relation is specified between siblings on the right hand-side, these are processed recursively and joined together.

5.7.1 Mapping Sets on States

The easiest part is finding which states to use in the STN. This is a one-to-one mapping of all enabled task sets which can be retrieved from the Task Model. So every $s \in E(M)$ is a state in the STN. For example; in the STN for the mobile phone example (figure 5.2(a)), each enabled task set is mapped on a state resulting in 4 different states.

5.7.2 Finding the Initial State

The initial state can be found by mapping the first enabled task set that will be presented to the user onto this state. This enabled task set is referred to as the *start enabled task set* or S_s. To find this enabled task set we first seek the left-most leaf t_l in the ConcurTaskTrees specification that is not a user task. This task appears before every enabling temporal operator so it must belong to the start task set.

However, t_l can belong to different enabled task sets when it has an ancestor involved in a concurrent temporal relation. If t_l only belongs to one enabled task set, the start enabled task set is found. To find S_s when t_l belongs to more than one enabled task set we check which tasks must belong to S_s by a recursive calculation of the first of the root. The enabled task set containing all the tasks of *firstTasks(root)* is selected. This enabled task set is unique, because the root node can not have ancestors involved in a concurrent relationship with its siblings.

Consider the example in figure 5.6(a). Taking the *firstTasks(root)* gives us $S_s = \{Task_{1.1}, Task_{2.1}\}$.

5.7.3 Detecting Transitions

Once all enabled task sets are mapped onto states of the STN, transitions between the different states have to be detected. Transitions are regular tasks; in [PS02] transition tasks between *Task Sets* are also defined but without letting the system detect them automatically. Our approach detects the transitions *automatically* relying on the temporal operators in the task model.

To detect candidate transition tasks, the task model has to be scanned for all candidate transitions according to definition 19. First, the task model is transformed into a priority tree before further processing. For every candidate task $t_1 \in \mathcal{T}(M)$ where $t_2 \overset{[>}{\rightarrow}_M t_1$ or $t_1 \overset{>>}{\rightarrow}_M t_2$ and t_1 and t_2 belong to differ-

ent enabled task sets [3], the selection of transition tasks out of the candidate transitions can be done as follows:

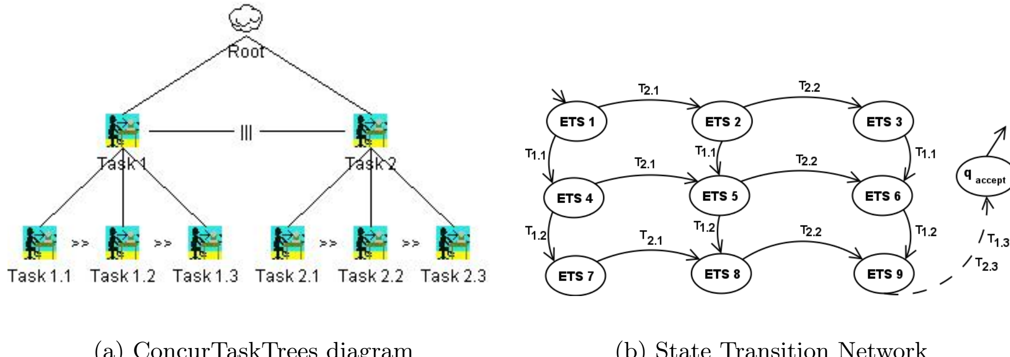
If the temporal operator is enabling: $t_1 \overset{>>}{\rightarrow}_M t_2$, one of the following four steps is taken:

1. t_1 and t_2 are leaves, one of the two following two steps is valid:
   (a) t_2 belongs to just one task set: all enabled task sets containing t_1 trigger the enabled task set that contains t_2.
   (b) t_2 belongs to several enabled task sets: for every e in $E(M), t_1 \in e$, there is a transition T and a task set T_l where T_l is the same task set as e except for t_1 is replaced by t_2 in the task set. Such a task set exists due to the presence of a concurrency temporal operator between ancestors of t_1 and t_2. Figure 5.6(b) shows the enabling transitions of the task model in figure 5.6(a). Consider the enabling relation between *Task 1.1* and *Task 1.2*. Three enabled task sets contain *Task 1.1* namely ets_1, ets_2 and ets_3 and three other enabled task sets contain *Task 1.2* namely ets_4, ets_5 and ets_6. ets_1 and ets_4 differ only by one task. Here *Task 1.1* is replaced by *Task 1.2* so we introduce this transition: $ets_1 \xrightarrow{Task1.1} ets_4$. All other transitions in this example can be found in the same way (e.g. ets_2 and ets_5 differ by the same tasks).

2. t_1 is a leaf, t_2 is not: t_1 triggers the enabled task set of the *firstTasks(t_2)* if t_1 and t_2 have no ancestor involved in a concurrent relation. Figure 5.7 shows how this situation maps on a STN. Even if one of the descendants of t_2 is involved in a concurrency or choice relation this does not change the process: they would belong to the same enabled task set by definition. If there is an ancestor of t_1 and t_2 which has a concurrent temporal relation with another task, detecting the correct transition is more difficult, and an approach similar as 1b is applied.

3. t_2 is a leaf, t_1 is not: in this case the triggering task is found by taking the right-most leaf of the descendants of t_1. This can be done by using the function *lastTasks*: the tasks returned by *lastTasks(t_1)* are the tasks that trigger t_2. Figure 5.8 shows an example. If there is an ancestor of t_1 and t_2 which has a concurrent temporal relation with another task, the same approach as 1b has to be applied.

[3] Notice t_1 and t_2 have the same ancestors, since they are siblings



(a) ConcurTaskTrees diagram (b) State Transition Network

Figure 5.6: ConcurTaskTrees with concurrency (5.6(a)) and the resulting State Network Diagram (5.6(b))

4. Neither t_1 nor t_2 are leaves: *lastTasks(t_1)* collects the "last" tasks of t_1 as in 3. Now apply 2 on each last task and t_2 as if they had an enabling temporal operator between each other. Figure 5.9 shows an example.

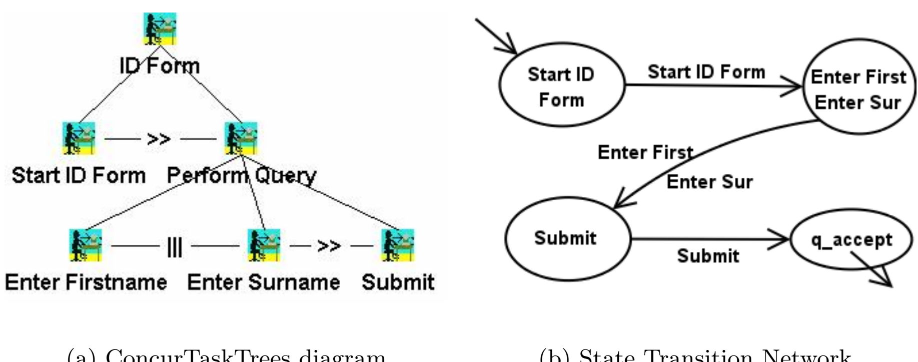
If the temporal operator is disabling: $t_1 \overset{[>}{\rightarrow}_M t_2$, then $first(t_2)$ is a *disabling task* (*first* is defined in section 5.6) and one of the following steps is taken:

1. If t_2 has an ancestor that is an iterative task t_i: for each enabled task set E containing t_2 add a transition $E \overset{t_2}{\rightarrow} startTaskSet(t_i)$, where *startTaskSet($t_i$)* is the first task set of the subtree with root t_i.

2. If t_2 has an ancestor a with an enabling operator on the right hand side: let r be the right hand side sibling of a and add transitions as if there is an enabling operator between t_2 and r.

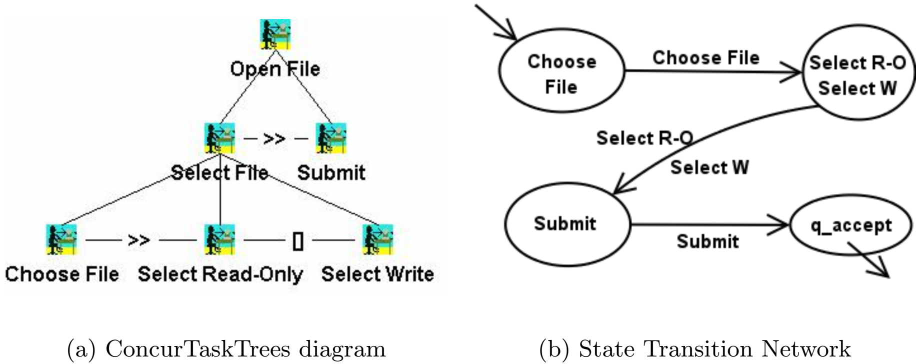
3. In all other cases; for each enabled task set E containing t_2 add a transition to the accept state: $E \overset{t_2}{\rightarrow} q_{accept}$. q_{accept} is a finishing state and will be further explained in section 5.7.4.

Figure 5.10 shows a ConcurTaskTrees specification with a disabling relation and the extracted STN. Notice how the task *Quit* is responsible for both transitions to the accept state.



(a) ConcurTaskTrees diagram (b) State Transition Network

Figure 5.7: $Task_{Start\ ID\ Form}$ is a leaf and $Task_{Perform\ Query}$ is no leaf.



(a) ConcurTaskTrees diagram (b) State Transition Network

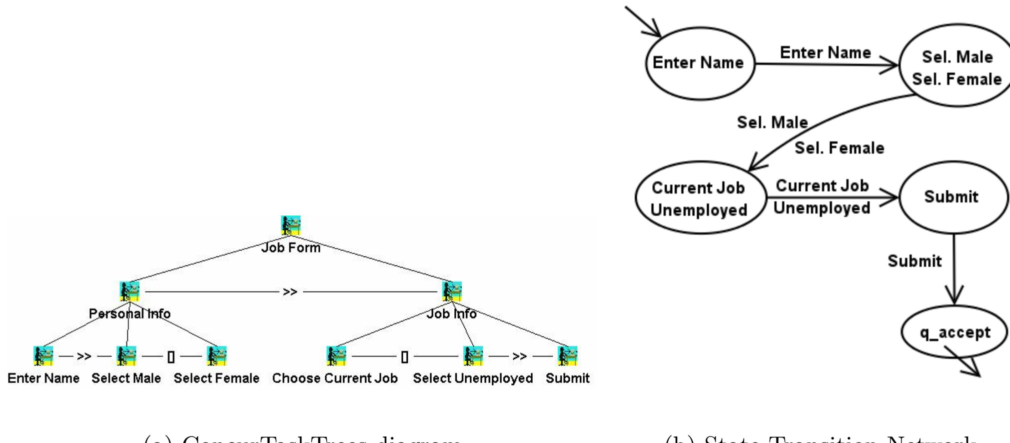
Figure 5.8: $Task_{Submit}$ is a leaf and $Task_{Select\ File}$ is no leaf.

5.7.4 Mapping the Finishing States

To complete the STN, the "last" enabled task sets with which the user interacts have to be located. For this reason we introduce a new definition for *expiring task*.

Definition 20 A task $t \in \mathcal{T}(M)$ is an **expiring task**, when t is a leaf and there is no $t' \in \mathcal{T}(M)$ such that $t \xrightarrow{o}_M t'$ with $o \in \mathcal{O}$.

If an expiring task t_e has an ancestor with an enabling operator on the right hand side, we have already taken care of this leaf by detecting enabling transitions (see section 5.7.3). If this is not the case, further examination of the



(a) ConcurTaskTrees diagram (b) State Transition Network

Figure 5.9: Neither $Task_{Personal\ Info}$ or $Task_{Job\ Info}$ are leaves.

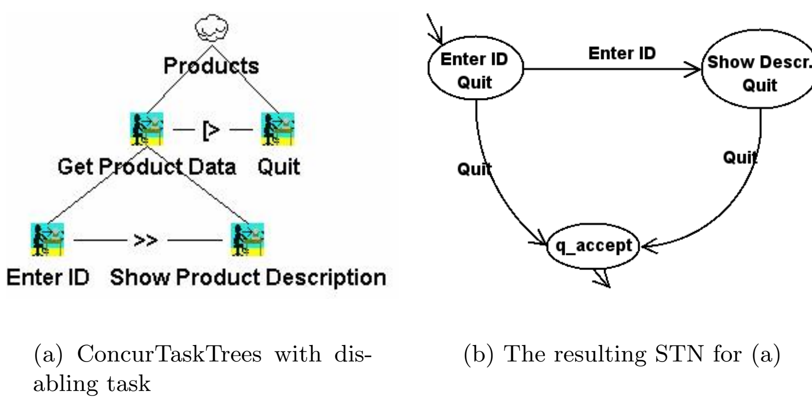
task is required:

- If t_e has an ancestor that is an iterative task t_i: for each enabled task set e containing t_e add a transition: $e \xrightarrow{t_e} startTaskSet(t_i)$.
- Else: for each enabled task set e containing t_i add a transition: $e \xrightarrow{t_e} q_{accept}$.

5.7.5 The resulting State Transition Network

Once the system has processed the steps described in the previous sections, a complete STN has been built. This STN describes a dialog model that is correct w.r.t. the task specification: the order of tasks that can be executed (the order between the enabled task set) is now also expressed in the STN. This is a powerful tool for designers to check whether their task specification meets the requirements before the working system has to be built. The designer can rely on the STN to produce a usable prototype supporting navigation through the user interface.

Referring back to the task specification for the email application shown in figure 5.4 and its enabled task sets shown in equation 5.2, we can construct the STN for this example by applying the algorithm from this chapter on the specification. The resulting STN can be seen in figure 5.11. Since the High-Level User Interface Description Language for each enabled task set is available, we



(a) ConcurTaskTrees with disabling task

(b) The resulting STN for (a)

Figure 5.10: Extracting the STN when a disabling relation is involved

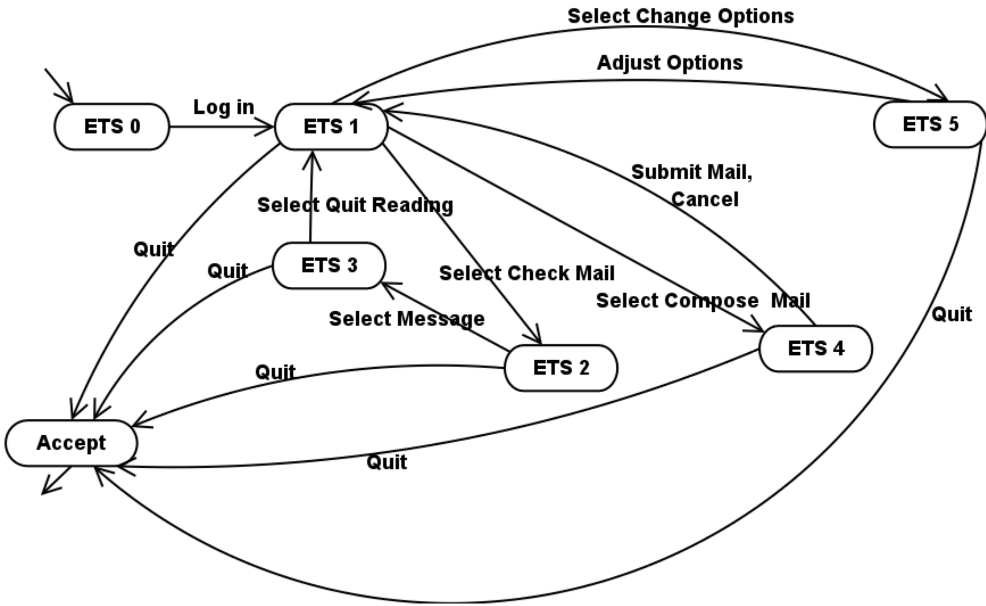


Figure 5.11: The state transition network for the email application

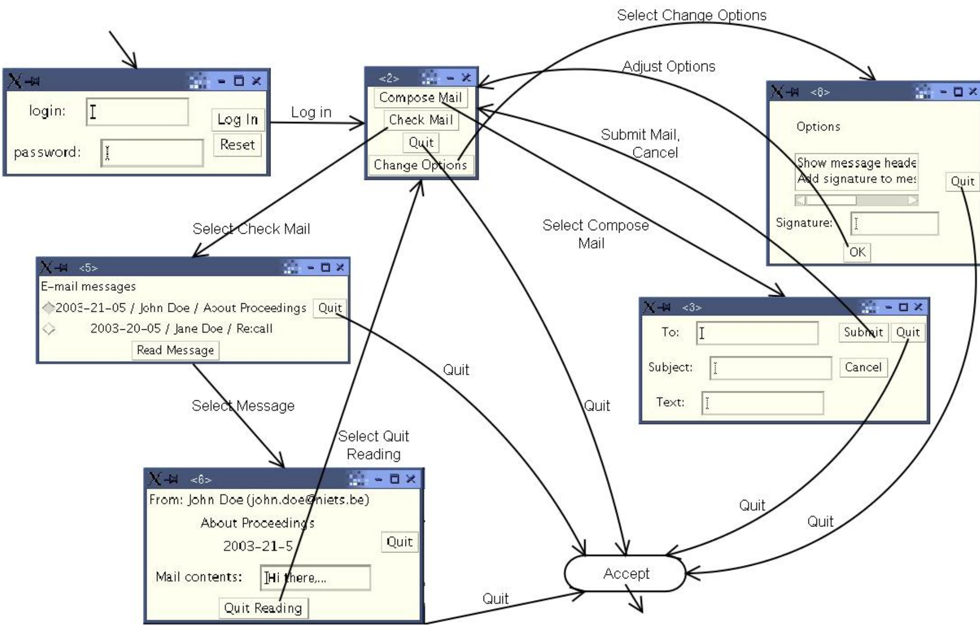


Figure 5.12: The state transition network for the email application

can render the concrete user interface for each state in the STN. Figure 5.12 shows the STN with each state replaced by its concrete user interface. The advantage of the Dygimes framework is it allows direct prototyping this way: the designer can see immediately how changes in the task specification reflect on the user interface compositions and user interface navigation.

5.8 Actual transitions between dialogs

Once the STN is completely defined, the system still lacks a way of detecting the actual conditions when the transition takes place. In most cases the condition for a transition is the execution of a task, so this has to be monitored during the lifetime of the application. The run-time system for executing the models is responsible for this monitoring role. In the Dygimes framework, where high-level XML-based user interface descriptions are attached as user interface building blocks to leaves in the task model, a specialized action handling mechanism [CLV+03] is implemented to take care of the state transitions. For now, widgets playing a key role for a dialog (e.g. a "next" button in an installation wizard) are identified by the designer by introducing a special

"action" and attaching it to the presentation model. The specialized action contains the preconditions and will be executed only when the preconditions are fulfilled.

When several concurrent tasks are included in the same enabled task set, which are all labeled as a transition in the STN, the system has to wait until all these tasks are finished. This can not be detected in the STN, in this case knowledge about the temporal relations is necessary. One possible solution is to group the concurrent tasks and handle them as if they were one task. For desktop systems concurrent tasks are not unusual, but for small devices (like PDA, mobile phone) the concurrency will be limited by the constraints of the device.

5.9 Discussion

This chapter showed how the task, dialog and presentation model are related to each other in the Dygimes user interface creation process. We believe the approach proposed here is a simple and intuitive approach in contrast with most other approaches. It is not bound to any notations, but does rely on the possibility of grouping tasks from the task model based on the temporal relations that are specified by the task specification. We will show how this approach is also applicable with other XML-based High-Level User Interface Description Languages that allow to define user interface structure and layout separately (see chapter 9).

An important advantage of our approach is it requires no completely new integrated tools to create the task specification or XML-based High-Level User Interface Description Language; it is applicable with the existing tools like the ConcurTaskTrees Environment for the task model and SEESCOA XML, XForms or UIML for the presentation model. Whenever the XML-based High-Level User Interface Description Language can be split up into several parts which can be randomly merged afterward, the same approach as presented in this chapter and chapter 4 can be used.

Chapter 6

Presentation of the User Interface

Contents

6.1 Introduction 95
6.2 Towards an XML-based HLUID Language 97
6.3 A Declarative Language for User Interface Design 98
6.4 SEESCOA XML 100
6.5 UiBuilder: A SEESCOA XML Renderer 105
6.6 Event handling in SEESCOA XML 110
6.7 Discussion 113

6.1 Introduction

The most visible and concrete model that can be found in Model-Based User Interface Development is the presentation model. It defines the actual user interface and is the last step before the final user interface. In the Dygimes framework the presentation model has support for four different aspects that are applicable for presentation models in general:

Structure : the structure is just an hierarchical view on the user interface, like shown in figure 6.1. This chapter will show more details about this aspect.

Layout : the layout for a multi-device presentation model should be flexible and more scalable than the layout management of traditional widget

sets. Because of its importance a separate chapter is devoted to layout management (chapter 7).

Rendering hints : the user interface needs to be tuned so it also fits the specific needs of the designer. Section 4.7 was devoted to this subject.

Widget mappings : to show a user interface, the structure containing abstract representations of the user interface has to be converted in a widget set specific user interface. The widget mappings convert the structure in an appropriate containment hierarchy using a set of concrete widgets that are defined in these mappings.

The information contained in these four parts of the presentation model should be sufficient to create a complete user interface. Notice the presentation model can be used as a stand-alone model: it does not depend on other concrete models like the dialog model for example. These four aspects presented here are *not unique* for the SEESCOA presentation model: other High-Level User Interface Description Languages also contain these four different parts (see chapter 3).

For reasons explained in sections 6.2 and 6.3 we choose to represent the presentation model as an XML-based language. This happened at the end of the year 2000, at that moment there were only few well-known XML-based languages in contrast with four years later. At the time SEESCOA XML was implemented, only UIML, XUL and XIML were well-known, and XForms was just starting. Our main target was somewhat different from the approaches that existed back then: to create a multi-device user interface language that was flexible enough to be used for deeply embedded devices that offered a limited screen space and non-traditional input facilities (e.g. touch screen, keypad instead of keyboard,. . .). Keeping this in mind, SEESCOA XML was one of the first XML-based High-Level User Interface Description Languages that was especially created for embedded systems and mobile computing devices although only little was made public at the time. The name “SEESCOA XML” for our markup language was introduced by others because of its association with the IWT SEESCOA project. Since initially this markup was being used to develop multi-device user interfaces for this project, we kept the name.

SEESCOA XML can be categorized as “intention-based” (as defined by AUIML, see also section 3.3.1): section 6.4 defines a set of abstract interaction objects that are partially related to the intention or type of interaction that is requested. The difficulty was to find a suitable balance between an abstraction

that is high enough to support multiple widget sets and modalities on the one hand, and a language that is expressive enough to describe common user interfaces.

This chapter is structured as follows: section 6.2 identifies the requirements of a User Interface Description Language. Next, in section 6.3 a motivation is provided for choosing XML as the User Interface Description Language. Section 6.4 describes the SEESCOA XML-based High-Level User Interface Description Language. The scalability of SEESCOA XML was tested by using this description in a multi-modal environment: [LLCR03] provides some insights w.r.t. this assessment of the language. The chapter is concluded with a discussion in section 6.7.

6.2 Towards an XML-based High-Level User Interface Description Language

Because of the evolving market towards embedded systems and mobile computing devices, a more general approach for describing a user interface for an embedded system or mobile computing device is necessary. The specification of the presentation model should not contain device- or platform-specific information because this prevents reuse of the specification. This is in fact the reoccurring theme in this dissertation text and the reason we describe the presentation model with a High-Level User Interface Description Language. In search of a notation for describing such a presentation model it should satisfy the following requirements, which we identified by experience :

Platform independent : because of the heterogeneity of embedded systems, a user interface designer should be allowed to design without having to worry about the system on which the interface will be used. Of course there are certain restrictions to this, which we will discuss further on in this text.

Declarative : describing a user interface asks for a certain level of expressiveness for describing human-computer interaction.

Consistent : the notation should be consistent and well-defined.

Unconventional I/O : embedded and mobile computing devices are less conservative in input and output devices. For example: while most contemporary desktop computers have a mouse and a keyboard, this

is not a requirement for a mobile device, which could very well have a touch-screen and speech recognition.

Rapid prototyping : in a highly competitive market, such as mobile devices, developers and designers want to tailor the software towards the users or user groups. A user interface notation should allow rapid prototyping to get the users involved in the development process sooner.

Constraint sensitive : because of the constraints embedded systems are coping with, the designers must be able to specify the constraints, or have the system automatically generate them.

Easily extensible : we want to extend our user interface with extra functionality, without starting from scratch.

Reusability : when a family of products is evolving, we want to reuse the design for the old devices in an optimal way.

Notice these are not style guidelines and have no influence on the actual appearance or usability of the interactive system. These guidelines are rather structural guidelines that define the requirements for a suitable specification language.

6.3 A Declarative Language for User Interface Design

The previous section listed several properties the user interface description language should have. Instead of creating a new kind of description language from scratch, we propose the usage of the eXtensible Markup Language (XML) for describing a user interface. This description language can offer us the properties we want:

Platform independent : XML is platform independent in the same sense that Java is platform independent: if there is an XML parser available on the system, the XML description can be used. If there is no suitable XML parser available for your target platform, XML is so simple that writing your own parser is fairly easy.

Declarative : one can specify "what" kind of interface is desired, without specifying "how" this interface should be built.

Consistent : through the usage of DTD[1] XML can be consistent. A DTD specifies a set of rules the XML file has to fulfill.

Unconventional I/O : XML can describe unconventional I/O: there are plenty of examples to provide evidence, e.g.: WML [Cov01] and VoiceXML [con01d].

Rapid prototyping : in various ways an XML-based user interface description can be rendered; e.g. with a stylesheet in a browser or with a Java-based XML renderer.

Constraint definitions : XML can contain constraint definitions; as well for the form of the XML itself, as for external resources we can add constraint definitions. An example language that is widely used is the Resource Description Framework (RDF, `http://www.w3.org/RDF/`) language.

Easily extensible : because XML is a metalanguage it is by nature an extensible language.

Reusability : it is relatively easy to reuse an existing piece of XML in a new design.

There is another advantage not addressed in the previous paragraph: because of the simple grammar and structure of XML it is an intuitive markup language. User interface designers do not need a firm technical background to work with XML. Also, it is easy to convert an XML description to different kinds of output presentations using XSLT[2]. Using XSLT, XML can be converted into HTML+CSS[3] for desktop browsers, into VoiceXML for speech driven input or into WML for mobile phones.

This is not the first time XML is proposed to be used as a user interface description language; chapter 3 discusses some other initiatives that are comparable with our approach. While most of these description languages only work at design time, we would like to propose an architecture for *run-time* user interface creation and adaptation, inspired by migratory applications [BC95] and remote user interface software environments [LK93]. This would enable us to "download" a user interface together with constraints and necessary transformations. Our description language should serve two purposes: adaptation

[1] Document Type Definition
[2] eXtensible Stylesheet Language Transformations
[3] Cascading StyleSheets: a stylesheet for an HTML document

and plasticity of user interfaces like introduced in [TC99]. While enabling us to tailor the user interface for particular devices and particular users (adaptation) it should take the defined constraints into account while preserving usability.

Having summarized the benefits of using XML as a User Interface Description Language, it remains an open issue how the user interface will be presented in the XML file, including the constraint definitions. Looking at figure 6.1, we see that a user interface can be structured as a tree, which is the basic structure of an XML file. We have a main window in which the user interface building blocks like buttons, sliders, etc. are laid out. In the main window we can have other windows containing building blocks, which in turn can be windows. It is advisable to make an abstraction here, like the proposed distinction into AIOs and CIOs [VB93], presented in figure 6.1, or to use abstract widgets [KAS96]. An AIO represents an abstract interface widget, independent of any platform. A CIO is a possible "implementation" for such an AIO. Using these concepts allows abstracting the user interface independent of the target platform. Abstract widgets represent practically the same thing: they are abstract platform independent representations of platform dependent widgets. If we want to add run-time layout management taking into account constraints defined by the environment, we will have to dynamically change the presentation of an AIO. This can happen due to screen size limitations for example. [EVP01] tries to solve this problem at design time using an intelligent agent (a mediator) for laying out user interface components.

For mobile devices this seems too much focused on actual screen-output, because no unconventional output device is taken into account. There might be an embedded system or computing device that has no screen at all, and has only some buttons and speech interaction for example. Then the on-screen data could be converted into a spoken dialog either way. Assuming speech interaction can be stored in XML, we could follow the same tree-structure for a speech-enabled dialog as we did for the windows.

6.4 SEESCOA XML

In this section we introduce the original Document Type Definition (DTD) for the SEESCOA XML language. First introduced in [LC01], it has evolved from a "serialization" language that was in syntax close to program objects into an abstract notation for specifying common user interfaces for multiple devices. SEESCOA XML is comparable with AUIML (see section 3.3.1 and [Mer01, MWK04]) as it tries to define abstract interaction objects. It can

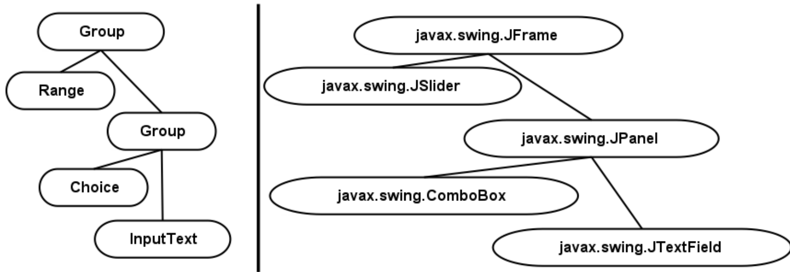


Figure 6.1: On the left a contextual representation (AIO), on the right the java.awt classes used to represent the presentation (CIO)

also be compared to the XForms notation, which is closest to the expressive power that can be found in SEESCOA XML for describing the user interface presentation [VLC04]. Although the SEESCOA XML was never released as an official User Interface Description Language, it was one of the first attempts to specify an XML-based High-Level User Interface Description Language targeted towards embedded systems (and later on mobile devices). It was designed to be compact and easy to write, interpret and render. The limited set of interactors one can specify emphasis this: button, label, textfield, range, choice, videowidget, table, canvas are the types of interactors that are supported. These interactors provide a clear notion of their purpose while a renderer can take advantage of the multiple possible presentations that can be provided for the interactor.

For readability purposes, the schema for the SEESCOA XML language is expressed as a DTD document in listing 6.1.

Listing 6.1: The SEESCOA High-Level User Interface Description Language Schema.

```
<!ELEMENT ui (title,group)>
<!ELEMENT title EMPTY>

<!ELEMENT group (interactor|group)*>
<!ATTLIST group name ID #REQUIRED
                            x CDATA #IMPLIED
                            y CDATA #IMPLIED
                            rows CDATA #IMPLIED
                            columns CDATA #IMPLIED>
```

```
<!ELEMENT interactor (button|label|textfield|range|
                             choice|videowidget|table|canvas)>
<!ATTLIST interactor x CDATA #IMPLIED
                           y CDATA #IMPLIED>

<!ELEMENT button (info,action?)>
<!ATTLIST button name ID #REQUIRED>

<!ELEMENT label (info)>
<!ATTLIST label name ID #REQUIRED>

<!ELEMENT textfield (info,text?,size)>
<!ATTLIST textfield name ID #REQUIRED>

<!ELEMENT range (info,min,max,start,tick,action?)>
<!ATTLIST range name ID #REQUIRED>

<!ELEMENT choice (info,choicetype,item+)>
<!ATTLIST choice name ID #REQUIRED>

<!ELEMENT videowidget (info,mediasource,withcontrols)>
<!ATTLIST videowidget name ID #REQUIRED>

<!ELEMENT table (info,rows,columns,columnnames,tablerow+)>
<!ATTLIST table name ID #REQUIRED>

<!ELEMENT canvas ( (info,height,width,graphic*)|(info,graphic))>
<!ATTLIST canvas name ID #REQUIRED>

<!ELEMENT info (#PCDATA)>
<!ELEMENT text (#PCDATA)>
<!ELEMENT size (#PCDATA)>
<!ELEMENT min (#PCDATA)>
<!ELEMENT max (#PCDATA)>
<!ELEMENT start (#PCDATA)>
<!ELEMENT tick (#PCDATA)>
<!ELEMENT choicetype (#PCDATA)>
<!ELEMENT item (#PCDATA)>
<!ELEMENT mediasource (#PCDATA)>
```

```
<!ELEMENT withcontrols (#PCDATA)>
<!ELEMENT rows (#PCDATA)>
<!ELEMENT columns (#PCDATA)>
<!ELEMENT columnsnames (#PCDATA)>
<!ELEMENT tablerow (#PCDATA)>
<!ELEMENT height (#PCDATA)>
<!ELEMENT width (#PCDATA)>
<!ELEMENT graphic (#PCDATA)>
<!ELEMENT action (ANY)>
<!ATTLIST action type CDATA #REQUIRED>
<!ELEMENT func (#PCDATA)>
<!ATTLIST func class CDATA #REQUIRED>
<!ELEMENT port (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT param EMPTY>
<!ATTLIST param name IDREF #REQUIRED>
<!ELEMENT update (#PCDATA)>
```

Listing 6.2 provides an example of how a user interface for a surveillance camera can be described in SEESCOA XML. Listing 6.2 is *not* simplified: the user interface description is meant to be human-readable and machine-processable at the same time. The description allows human users to specify the user interface on a high level. On the other hand, the structured and hierarchical approach by using XML as a notational language to describe the user interface allows machines to process and use these descriptions without human intervention. Our notation uses a range of tags that are easy to read and understand for humans. Care has been taken to introduce no ambiguities in the specification and to enable easy migration to other specification languages.

Listing 6.2: An example SEESCOA XML listing for a camera. Developed for the SEESCOA researcht project (IWT 980374) in cooperation with other partners.

```
<ui>
<title>Camera</title>
<group name="videopanel">
  <interactor>
    <video name="video">
      <text>Camera 2 video stream</text>
      <mediasource>http://twiki.luc.ac.be/camera:8888</mediasource>
```

```
    </video>
  </interactor>
  <interactor>
    <range name="zoomrange">
      <text>Zoom</text>
      <min>-100</min>
      <max>100</max>
      <start>0</start>
      <tick>25</tick>
      <action>
        <func service="Mosaic.camera2">setZoom</func>
        <param name="zoomrange"/>
      </action>
   </range>
  </interactor>
  <interactor>
    <range name="focusrange">
      <text>Focus</text>
      ...
      <action>
      <func service="Mosaic.camera2">setFocus</func>
        <param name="focusrange"/>
      </action>
    </range>
  </interactor>
</group>
</ui>
```

The available types of tags are limited, but a lot of dialog-based user interfaces can already be implemented using these widgets (e.g. all kinds of web forms). There are two tag types which are of particular importance: **group** tags and **action** tags. The **group** tags allow to group objects which have no meaning when they are separated . An example of this is a "date interactor": the interactors involved for filling in a date should not be separated (listing 6.3). Groups can be nested: they can be hierarchically structured. This enables us to reuse groups of interactors, and make new composed groups. The **action** tags allow a user to specify which action to fire if the interactor (which is the parent node) is manipulated. The action tag specifies the target (this can be a class name, a server,. . .) and the functionality that has to be invoked

from this target. It is also possible to specify parameters and use the names of the interactors or groups for these parameters. Our system will automatically extract the current content out of the interactor or group (to which these parameter identifiers point) and pass it to the invoked functionality. Section 6.6 provides more insight in the use of the action element: it is used to link the user interface to "domain" or "application" objects.

Listing 6.3: A date group

```
<group name="date">
 <interactor>
  <range name="day">...</range>
 </interactor>
 <interactor>
  <range name="month">...</range>
 </interactor>
 <interactor>
  <range name="year">...</range>
 </interactor>
</group>
```

6.5 UiBuilder: A SEESCOA XML Renderer

Since an XML-based User Interface Description Language needs a renderer or a transformer to generate a concrete user interface from an XML document, SEESCOA XML is supported by the rendering engine *UiBuilder*. UiBuilder is part of the Dygimes system, and can create a concrete user interface from a SEESCOA XML document in Java AWT, Java Swing, Java kAWT and HTML. For parsing the XML document, both the DOM[con01b] and SAX[4] interfaces can be used by the renderer to ensure maximum portability. The UiBuilder rendering engine is limited to providing a translation between the abstract and concrete user interface and only takes care of singular presentation units.

Figure 6.2 shows an interface for a camera surveillance system, where two SEESCOA XML documents are merged to obtain the concrete interface. Part (1) is the user interface for a camera component and visualizes only the controls of the camera (see also listing 6.2). Part (2) visualizes a motion detection component that could be used by the surveillance system. Both components

[4]SAX: Simple API for XML

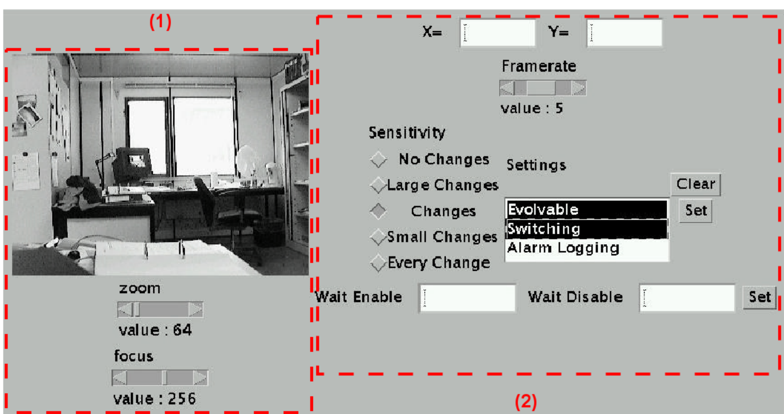


Figure 6.2: Camera-based surveillance system using UiBuilder to render a user interface for the camera [RB03] : part (1) shows the interface for listing 6.2, part (2) shows the interface for listing 6.4.

.

provide their own SEESCOA XML description (listing 6.4 shows part of the complete document). UiBuilder merges the various SEESCOA XML document to present a single user interface. Chapter 8 elaborates on the relation between software components and SEESCOA XML.

Listing 6.4: SEESCOA XML description for a motion detection software component. Developed for the SEESCOA researcht project (IWT 980374) in cooperation with other partners.

```
<ui>
 <title>Motion Detector</title>
 <group name="mgui" rows="2" columns="1">
       <group name="canvaspointer" x="1" y="1"
                           rows="1" columns="1">
              <group name="coordinatesoutput" x="1" y="1"
                                          rows="1" columns="2">
                     <interactor x="1" y="1">
                            <textfield name="pX">
                                   <info>X=</info>
                                   <text>     </text>
                            </textfield>
                     </interactor>
```

```
                <interactor x="2" y="1">
                        <textfield name="pY">
                                <info>Y=</info>
                                <text>     </text>
                        </textfield>
                </interactor>
        </group>
</group>
<group name="settings" x="1" y="2"
                        rows="3" columns="1">
        <interactor x="1" y="1">
                <range name="framerate">
                        <info>Framerate</info>
                        <min>1</min>
                        <max>10</max>
                        <start>5</start>
                        <tick>1</tick>
                        <action type="component">
                        <func class="testcases.scss.MDClient">
                        GUISet</func>
                        <param name="framerate"/>
                        <type>framerate</type>
                        <port>EventsMD</port>
                        </action>
                </range>
        </interactor>
        <group name="osettings" x="1" y="2"
                        rows="1" columns="3">
                <interactor x="1" y="1">
                        <choice name="sens">
                                <info>Sensitivity</info>
                                <choicetype>single</choicetype>
                                <item>No Changes</item>
                                <item>Large Changes</item>
                                <item>Changes</item>
                                <item>Small Changes</item>
                                <item>Every Change</item>
                        </choice>
                </interactor>
```

```
                <!-- ... -->
                </group>
        </group>
</group>
</ui>
```

Figure 6.3 gives an overview of the core class hierarchy of the UiBuilder rendering engine. Notice it has a hard-coded set of classes of the interactors that can be used. In contrast with the UIML renderer presented in chapter 9, UiBuilder is not extensible to other widget sets or new widgets without writing new code. The primary reason was the original target for the UiBuilder rendering engine: the main deployment platform were (custom) embedded systems. Figure 1.1 on page 6 shows a setup with some custom hardware that uses the UiBuilder rendering engine to generate the user interface.

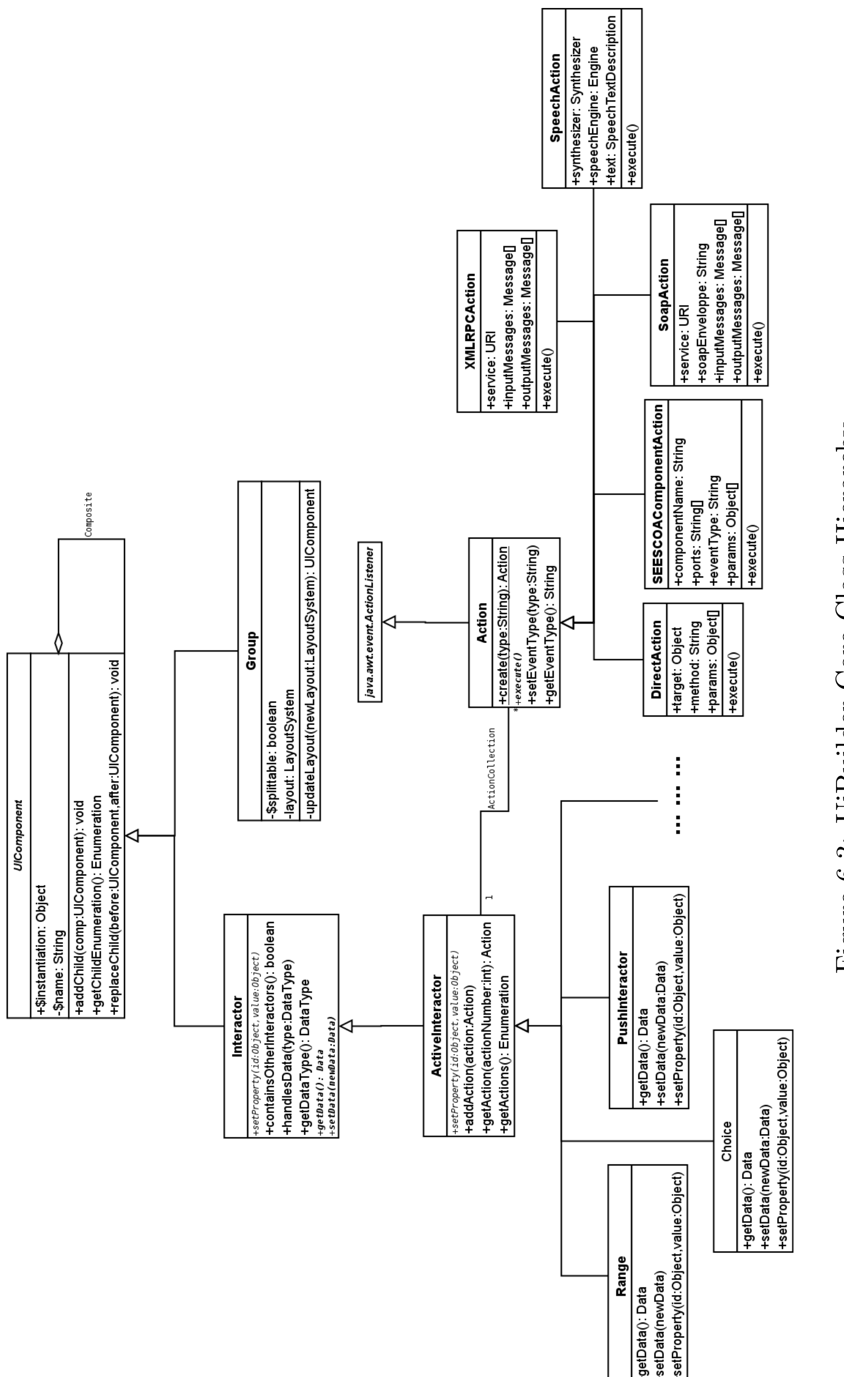


Figure 6.3: UiBuilder Core Class Hierarchy

The layout of a user interface is supported in two ways by the renderer: a grid-based layout mechanism and a constraint-based one. The former is simple and has a predictable effect, while the latter is far more scalable but less predictable. The constraint-based layout mechanism is discussed in chapter 7. The grid-based layout can be used by adding two attributes to each group and interactor: the x- and y-coordinates. If no placement is provided, the layout manager will place all interactors along a vertical line following the order of the interactor specification in the XML document.

6.6 Event handling in SEESCOA XML

In SEESCOA XML it is very simple to associate an action with an event of a widget. We take advantage of the XML structure by adding `action` tags as children of interactors. Section 4.5 showed the action tags are part of the interaction model provided by Dygimes. SEESCOA XML offers an extensible action mechanism, that allows to have a custom description of the application binding. The sole assumption is that the SEESCOA XML renderer should implement the plugin for the type of action that is provided. Figure 6.3 shows the place of the action system in the object hierarchy of the Uibuilder core. This class diagram is simplified to give an overview of the core code architecture.

In the UiBuilder core implementation, every class that inherits from the `Action` class implements a specific protocol. The pieces of implementation that support an action protocol are referred to as "action plugins". The designer or user interface developer has to specify which type of action is specified between `<action type="thetype">` and `</action>`. While parsing a SEESCOA XML document, UiBuilder will load the appropriate action plugin and pass the action subtree so it can be processed and stored by an action plugin object. An action is an "executable" object: it has an execute funtion that performs the statements specified in the subtree of the action element. This can include information with application objects, executing scripts (Python scripts are supported) or even creating a speech-based response to an event.

Listing 6.5 shows an example of two range interactors that communicate with each other by using direct method invocation (DMI) on existing objects. There is an update tag that can specify the interactors that consume the result of an action if there is a result available. In this case the implementation of `examples.Default.echo` merely passes the value, which it gets from the parameter `<param name="range1"/>`, back as a result. In the example both ranges will always show the same value.

Listing 6.5: Communicating range interactors in SEESCOA XML

```
<ui>
<title>Range example</title>
<group name="ranges" rows="1" columns="2">
       <interactor x="1" y="1">
              <range name="range1">
                     <min>0</min>
                     <max>1000</max>
                     <tick>1</tick>
                  <action>
                    <func class="examples.Default">echo</func>
                    <param name="range1"/>
                    <update>range2</update>
                   </action>
              </range>
       </interactor>
       <interactor x="2" y="1">
              <range name="range2">
                     <min>0</min>
                     <max>1000</max>
                     <tick>1</tick>
                  <action type="direct">
                     <func class="examples.Default">echo</func>
                     <param name="range2"/>
                     <update>range1</update>
                    </action>
              </range>
       </interactor>
</group>
```

Listing 6.6 shows the simple SEESCOA XML description for the user interface in figure 6.5 that copies text from one textfield into another one with only a small Python script. Jython[5] is used as Python interpreter, and the Python scripts could get data from the user interface that was rendered by using a special API.

Listing 6.6: Python support in SEESCOA XML

```
<ui>
```

[5] http://www.jython.org/

(a) AWT (b) Swing

Figure 6.4: Communicating range interactors rendered from a SEESCOA XML description

```
    <title>Simple Script Example</title>
    <group name="scripters-group">
            <interactor>
                    <textfield name="sourcefield">
                            <info>sourcefield</info>
                            <text>sourcetekst</text>
                    </textfield>
            </interactor>
            <interactor>
                    <textfield name="targetfield">
                            <info>targetfield</info>
                            <text>         </text>
                    </textfield>
            </interactor>
            <interactor>
                    <button name="script-it">
                    <info>Copy Text</info>
                    <action type="script">
                            <script type="python">
                                    <![CDATA[
from uibuilder.widgets import WidgetDataInPython
widgetdata = WidgetDataInPython()
textfieldData = widgetdata.getTextfieldData("sourcefield")
widgetdata.updateTextfield("targetfield",textfieldData)
                                    ]]>
                           </script>
                    </action>
```

```
                    </button>
            </interactor>
    </group>
</ui>
```

Figure 6.5: User Interface with Python support rendered in AWT from listing 6.6

6.7 Discussion

This chapter presented SEESCOA XML, an XML-based User Interface Description Language that was conceived in 2000 and which we mainly created in 2001 for embedded systems and mobile computing devices. A clear motivation why XML was chosen as a basis to create this language is provided. There weren't many other initiatives besides UIML, XIML, XForms and XUL at the time we created the SEESCOA XML language. In contrast with these other approaches, SEESCOA XML targets a renderer with a smaller footprint. In contrast with the current state-of-the-art in XML-based User Interface Description Languages as presented in chapter 3, the SEESCOA XML language already implemented several ideas that can be found among the other initiatives. This indicates the choices made when we started with SEESCOA XML were the correct ones. There is also a clear focus to support the presentation model; the other models in Dygimes are supported in another independent way (which will be shown in section 9.7).

This renderer, UiBuilder, is the part of Dygimes that converts an abstract user interface into a concrete user interface. One of the differences with other similar renderers is the support for custom protocols to interact with the application domain. UiBuilder has an easy extensible mechanism that supports different types of code execution mechanisms: Direct Method Invocation, XML-RPC, Python scripts, SOAP messages,. . . . There is no restriction on the type of protocol in the SEESCOA XML DTD.

The major drawback in our implementation is the predefined set of interactors. We can create simple form-based applications for a wide range of devices because we define a common denominator of the widgets supported

on these devices. This makes the user interface very portable, but much less expressive. In the discussion of chapter 3 we positioned SEESCOA XML (see also figure 3.1) as one of the XML-based User Interface Description Languages with the lowest user interface coverage. Although this means it is not capable of rendering "rich" graphical user interfaces, it does supports a large range of platforms without needing to change anything in the presentation specification. We started to create a state-of-the-art UIML renderer on the .Net Framework, presented in chapter 9, that provides an extensible architecture and is able to describe much richer graphical user interfaces then SEESCOA XML, but does not support the same degree of plasticity of a user interface without the need to change the presentation specification.

Chapter 7

Multi-device Layout Management

Contents

7.1 Introduction **115**
7.2 Related Work **116**
7.3 Constraint Satisfaction and Layout Management . **118**
7.4 Calculating Presentation Structures **120**
7.4.1 Describing spatial constraints 120
7.4.2 Building the layout description graph 121
7.4.3 Calculating widget positions 121
7.4.4 Conflict handling 122
7.4.5 Further screen space reduction strategies 123
7.5 Discussion **123**

7.1 Introduction

This chapter describes how a constraint-based layout management system enables the user interface designer to deploy his/her interface to a wide range of devices. A layout management system takes care of positioning components within the user interface. While the variety of mobile computing systems grows, the development techniques for providing user interfaces targeted at these systems are maturing slowly. Because of the many available programming languages and interface toolkits for creating user interfaces for mobile systems, there is a need to design the user interface independent of these

choices. On the other hand the approach should be practical to reduce the time-to-market.

Based on the different models we introduced in chapter 2, we define a layout management system that applies on *presentation units*. Since a presentation unit can be dynamically composed out of different tasks, the layout management system should be able to work on a user interface that was not explicitly designed but algorithmically composed. Since we can define building blocks in a High-Level User Interface Description Language, we can extend this High-Level User Interface Description Language with a specification for the layout of the user interface. Furthermore we need to apply this specification method recursively upward so different user interface description fragements can be composed to make up one presentation unit and have some kind of layout specification to relate the different parts in the composition w.r.t each other. The layout specification is created *at design-time*, while the actual layout of the user interface is computed *at run-time* according to the device constraints.

The next section gives an overview of the related work for this topic. Section 7.3 will discuss the usage of constraint solving and logical grouping of the layout manager we implemented [LCC03]. Details about the actual layout adaption process taking into account available screen space are explained in section 7.4. This chapter is concluded with a discussion of the obtained results in section 7.5.

7.2 Related Work

As we will show in section 7.4.1 the specification of the user interface structure and the specification of the layout can be separated. Since we target multiple devices with different resources like screen size and different widget sets, this separation is essential. There has been a wide diversity of work investigating appropriate layout algorithms for different kinds of computer-aided visualization of information.

Most noticeable are the existing layout managers like those used by the "classical" widget sets like Java AWT and Swing graphical user interface (GUI) toolkits [WC] or the Gimp Toolkit (Gtk) [MT02]. They provide an hierarchical approach, where layouts can be nested in other layouts. This has proven to be both intuitive and flexible. For example, Java-based GUIs can already be shown on different devices whilst adapting to their new environment. Unfortunately this approach lacks flexibility when the constraints of the new environment become more extreme. The GUI does not have the possibility to "regroup" itself in another presentation structure for better presentation while

respecting the constraints. In other words the current layout managers are not scalable enough to handle the wide difference in screen sizes from today's devices. Since it is even more difficult to obtain a layout management algorithm which scales over multiple modalities we will limit ourselves to graphical layout management. E.g. when we have an XML-based High-Level User Interface Description Language that can be converted into a form-based as well as a speech-based interface the layout should minimally specify spatial constraints for the form-based interface and temporal constraints for the speech-based interface. Since a general layout management system, that is independent of the concrete interface representation, remains insufficient for current emerging technologies we focus on finding a solution for graphical interfaces first. We believe there should be a different layout mechanism for each modality (e.g. graphical user interface of speech-based) that can be used since different modalities will have different notions of layout.

[LF01] provides a survey of work on automated layout techniques for presentation models. Most of the research on automated layout management is concentrated on constraint-based layout systems. Two approaches can be identified that are of interest for our work: *spatial layout constraints* and *abstract constraints*. Spatial constraints express the positioning of components with respect to each other. Abstract constraints express a logical relation between components. For example, "caption A is left of list B" is a spatial constraint and "caption A describes the content of list B" is an abstract constraint. Abstract constraints are always transformed into spatial constraints before they can be used.

[BHLV94] and [VG94, Van95] describe some techniques for automated layout management, more in particular the dynamic right-bottom and the static two-column strategy. These techniques are not constraint-based, as opposed to the approach we present in this chapter. In the past, there have been several attempts to create constraint-based layout managers, like models created in Thinglab [Bor79, MBFB89] and the application of the DeltaBlue algorithm in [SMFBB93]. Our approach is partially inspired by the work presented in [SMFBB93]. Two-dimensional GUIs are often described with linear constraints. Because constraint satisfaction is a difficult problem to solve, some research was conducted toward efficient algorithms for constraint satisfaction problems using linear constraints [BMSX97]. When transporting a GUI to another device with different constraints, a suitable presentation model has to be generated that is still conform with the predefined requirements for laying out the interface.

[EVP01] describes a method to generate and select a presentation model for

GUIs on mobile computing devices. It also describes the influence of interactor selection on the allocated screen space. Through iterative specialization steps a platform-optimized presentation can be retrieved from a platform independent user interface description. Each iteration can apply two redesign strategies in this approach: (1) remodeling logical windows within a presentation unit and (2) remodel AIOs in a logical window. Automatic interactor selection based on selection rules is described in [VB93]. Its purpose is to select an optimal set of Abstract Interaction Objects (AIOs) presenting some functionality for a specific target platform. Notice the selection rules from [VB93] can be used in the remodeling steps in [EVP01] to obtain more intelligent *and* adjustable adaptation behavior.

A more recent approach is SUPPLE [GW04] which treats the rendering of a user interface as an optimization problem. It takes into account a set of interface and device constraints to obtain a legal mapping from an interface element to a widget. Notice this is a one-on-one mapping from an AIO onto an individual widget. Notice this approach could also be combined with the selection rules of [VB93] to make it more adjustable for the designer.

7.3 Constraint Satisfaction and Layout Management

As a first step toward constraint-based layout management, we use four simple linear spatial constraints to describe the positioning of the widgets in respect to each other: *left-of*, *right-of*, *above* and *below*. In addition the available space to lay out the widgets is divided in a grid. Each bucket in the grid is uniquely identified by its x and y position within the grid. Notice the linear constraints can be expressed in a mathematical form: take the constraint *widget A left-of widget B* for example. If widget A is put in bucket X with coordinates (x_1, y_1) and widget B is placed in bucket Y with coordinates (x_2, y_2) than the constraint can simply be expressed as $x_1 < x_2$.

Several traditional layout managers do offer some kind of spatial constraints, like the GridBagLayout of the Java GUI toolkit[WC] or the "packing" containers provided by Gtk [MT02]. However, our approach differs with traditional approaches because we also use the *hierarchy* as described by the High-Level User Interface Description Language (see chapters 3 and 6) instead of directly implementing the hierarchy in the programming code. Constraints are only defined between siblings in the description tree: this means spatial constraints are only specified between abstract interaction objects in the same group or groups who share the same parent group in the hierarchy. A set of constraints is related to a group of widgets that logically belongs together

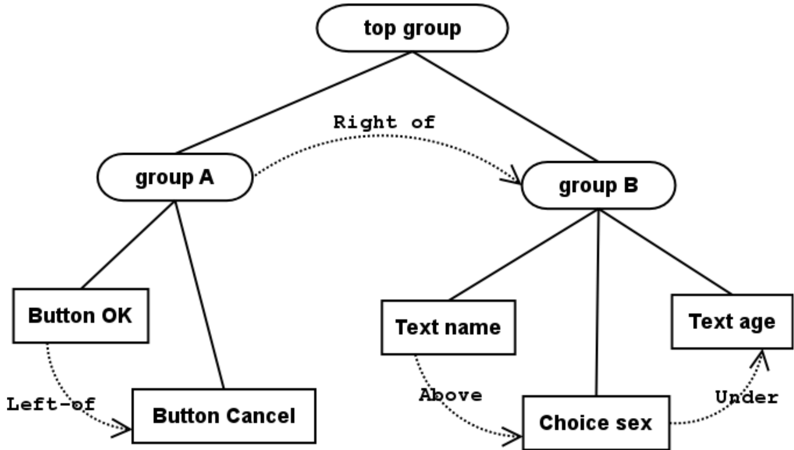


Figure 7.1: A visual representation of the constraint definition

this way. This is shown in figure 7.1. The hierarchy divides the interface in logical groups. These groups can be subdivided in other groups and so on. All widgets, part of the same group, have a logical relation with respect to each other. Some rules can be applied here:

- A group describes a set of **logically related** abstract interactors or groups of abstract interactors. The designer should decide which widgets are gathered in a group.

- A group can be specified **splittable**. This specifier allows the layout manager to show the abstract interactors or groups of abstract interactors in separate spaces.

- The group specifier **non-splittable** forces the layout manager to show the children of the group as a whole to make sense to the user. Notice non-splittable is only valid for the direct children of the group, and does not constrain the further offspring.

In contrast with the traditional layout managers, our system *does not rely* on a particular programming language to produce the GUI. In combination with the High-Level User Interface Description Languages introduced in chapter 6, we get a *very loosely-coupled* user interface for an application.

7.4 Calculating Presentation Structures

As mentioned before, only spatial constraints (specified at design-time) will be employed in our implemented system. The use of only this kind of constraints is sufficient because they directly determine the geometric structure of the layout. Abstract constraints which describe a high-level relation between two components are omitted in our approach because they are always transformed to spatial constraints in a later stage.

7.4.1 Describing spatial constraints

A simple XML-based syntax is used for describing constraints between two components. A formulation of a constraint network for listing 6.3 is depicted in listing 7.1. The first constraint in this network implies that *label* will appear somewhere above *date* in the resulting layout. If the set of constraints is sufficiently large, there is a strong likelihood that conflicts will arise: for example, some constraints may contradict others and possibly make the set of constraints unsolvable. For handling these kind of inconsistencies, priorities are introduced in our system. A priority, represented by an integer value, can be applied to each constraint. Higher values indicate stronger constraints and lower values represent weaker constraints.

Listing 7.1: A SEESCOA XML constraint description for a group of AIOs.

```
<constraints>
  <constraint type="above" priority="5">
    <interactor name="label"/>
    <interactor name="date"/>
  </constraint>
  <constraint type="left" priority="7">
    <interactor name="day"/>
    <interactor name="month"/>
  </constraint>
  <constraint type="right" priority="7">
    <interactor name="year"/>
    <interactor name="month"/>
  </constraint>
  <constraint type="left" priority="7">
    <interactor name="day"/>
    <interactor name="year"/>
```

```
  </constraint>
</constraints>
```

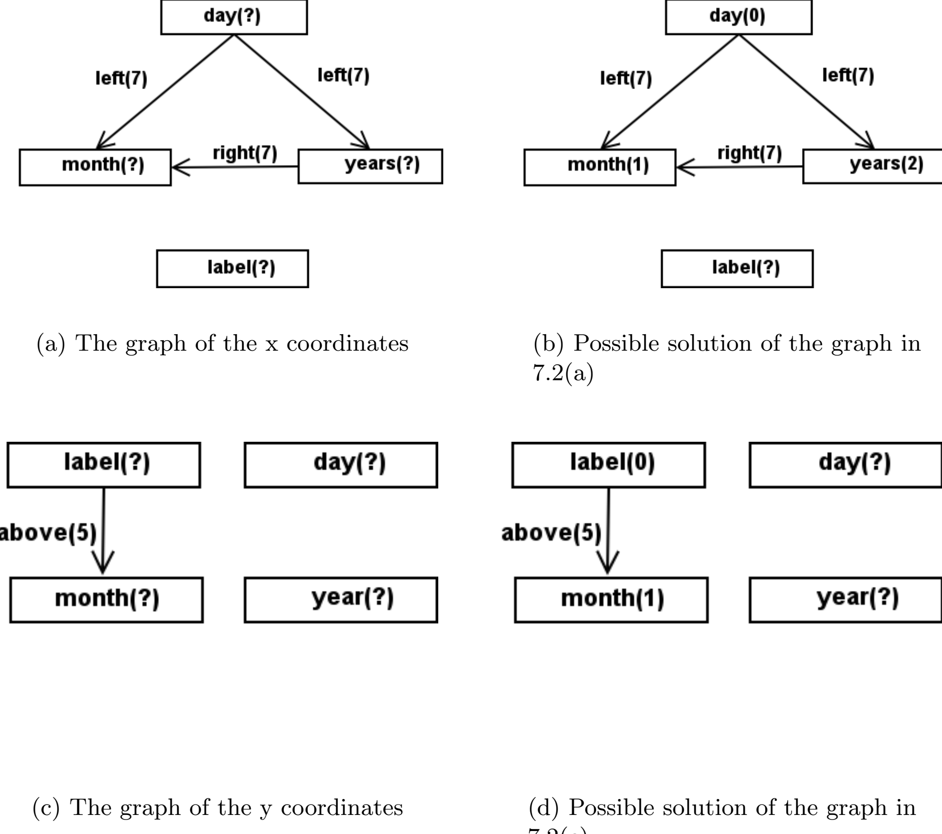
Once the layout specification is created, a concrete layout is computed *at run-time*. The user interface renderer will use the steps described in the following sections to find a suitable layout that respect the spatial layout constraints as described by the layout specification.

7.4.2 Building the layout description graph

Because constraints are only allowed between siblings, each component of the group will be associated by a unique bucket in the grid. Our implemented algorithm will initially solve the x coordinates of the components (coordinates indicate the position in the grid, *not* the absolute coordinates on the screen). A graph will be composed where each node represents a component with his x coordinate and each edge represents a constraint between the two components connected by the edge. Considering the constraints in listing 7.1, the resulting graph will have an outcome as depicted in figure 7.2(a). Like mentioned earlier, each edge with the label *left* can be expressed as $x_1 < x_2$. A possible solution for the constraints represented in the graph is depicted in figure 7.2(b). Single nodes represent components from which the x coordinates are not affected by the constraints. The y coordinates for these components will be determined at a later stage in the algorithm. After the solution has been calculated for the x coordinates, the same strategy will be applied for the y coordinates. The resulting graph and a possible solution for the y coordinates are depicted in figure 7.2(c) and figure 7.2(d).

7.4.3 Calculating widget positions

The results of the two proceeding steps will be combined. This leads to a general solution where the component *month* has the coordinates (1,1) for example. The final stage of the algorithm consists of assigning a value from the predefined domain to the still unassigned coordinate variables. The domain is the range of integers between 0 and the maximum cells plus one. When the component *day* with coordinates (?,0) is considered, the user interface designer has the responsibility to locate a free bucket in column 0 to place the component in. '?' reflects that every free place in column 0 can be used. This can be filled in automatically when required or can be chosen by the designer. E.g. the algorithm can choose the first bucket (=row) that is available in that column, just like a FlowLayout in Java would do.



(a) The graph of the x coordinates

(b) Possible solution of the graph in 7.2(a)

(c) The graph of the y coordinates

(d) Possible solution of the graph in 7.2(c)

Figure 7.2: The calculation of the presentation structure

7.4.4 Conflict handling

The occurrence of cycles in the graph implies the presence of conflicting constraints. Each edge in a cycle represents a constraint that conflicts with another one in the cycle. The constraint with the weakest priority will be removed to break the cycle. If the constraints have equal priority, the first one will be removed and the other will be maintained. The presence of multiple edges between two nodes also implies a conflict in the spatial layout constraints: a component can not be placed at the same time at the right and left side of another component. The edge with the highest priority will be kept,

and the conflicting constraints with lower priorities will be removed from the constraint graph. Notice this is still what the designer required, since it is the designers responsibility to specify the priority of the constraints.

7.4.5 Further screen space reduction strategies

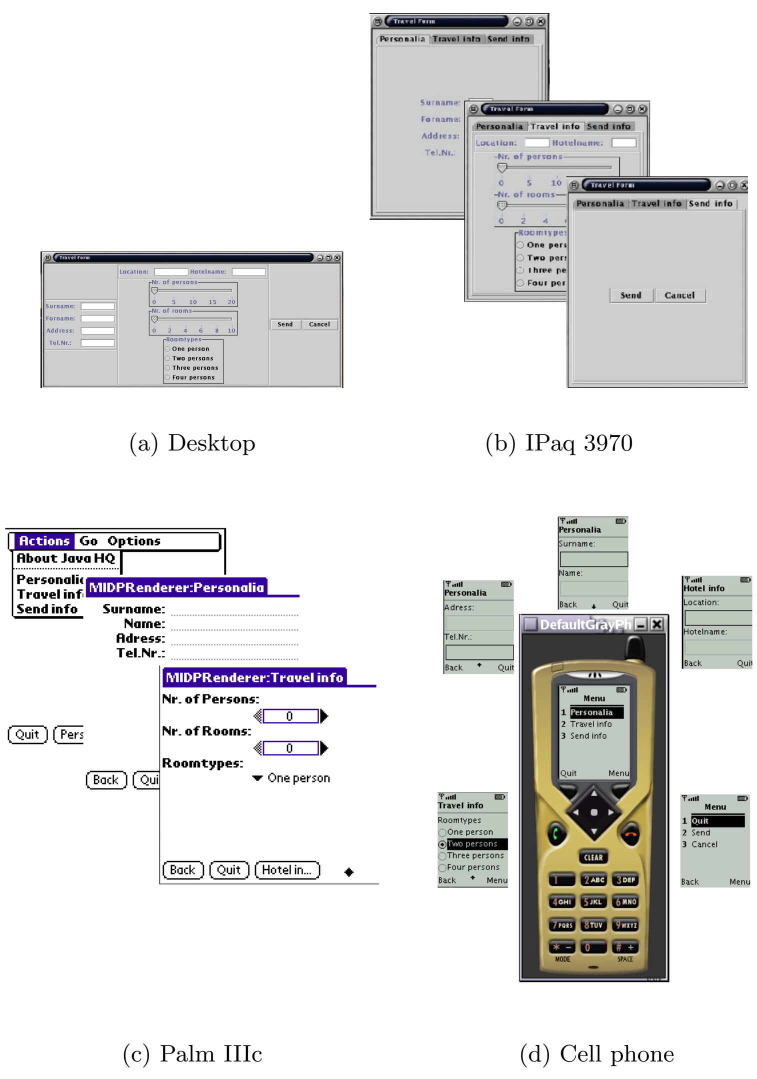
After the presentation structure which is defined by the user interface designer is calculated, the possibility exists that the layout does not fit on the screen. A layout adapter is imposed to resolve this problem by adapting the presentation structure to the limited screen size of the target platform. One of the strategies employed by the adapter to shrink the presentation consists of reducing the sizes of the textfields. Several similar rules can be applied (e.g. figures can be reduced). The functionality of the user interface will remain intact.

A more powerful strategy to reduce screen space consists of placing the components of a splittable group after each other in a card layout or with tabbed panes, like shown in figure 7.3. This strategy takes benefit from the hierarchical nature of the user interface specification. It is important to notice that after applying this method the constraints between the children of the group are no longer valid. The adapter applies these strategies iteratively on the presentation structure and after each iteration will be checked if the layout fits the screen. However sometimes it is impossible to shrink the layout to the size of the screen of the target platform. An appropriate warning will be displayed if the user interface cannot be rendered.

7.5 Discussion

The approach presented here is based on a set of simple spatial constraints to specify the layout of a user interface. Together with the hierarchy of the user interface they allow us to provide a flexible layout management system that scales from mobile phones towards desktop computers. There are several drawbacks in our approach however:

- It is only suitable for purely graphical two-dimensional user interfaces
- The designer is given only a limited overview the effects of the constraints that are specified because the user interface is dynamically composed by the executable models
- The system offers only limited expressibility in favor of a more scalable approach



(a) Desktop (b) IPaq 3970

(c) Palm IIIc (d) Cell phone

Figure 7.3: A multi-device hotel registration form

Being introduced in Dygimes at early 2003 it was a very experimental system that scaled from screens of desktop PCs to the screens of mobile phones like figure 7.3 shows. The algorithm used is comparable to the one introduced in RIML [KWWZ04] because this could be expressed in terms of presentation units (dialogs), hierarchical grouping with the “splittable” aspect and the simple spatial layout that use an underlying grid to calculate the logical position of the different elements. A graphical tool is available that allows to specify constraints in an interactive way. The graphical tool can render the user interface at any time so the designer can see the effect of adding,removing or changing a constraint. This tool is shown in figure 4.7 on page 67.

At the current stage we are working on a system that allows to distribute the interface over several devices according to the task, dialog and presentation specification [VC04]. One step in this direction are migratable user interfaces according to the context-of-use. The devices that are available in the user's environment can be part of this context-of-use. Our current extension, DynaMo-AID, written on top of the Dygimes framework supports exactly this and will be presented in chapter 10.

Chapter 8

Components and Multi-device User Interfaces: The SEESCOA experiment

Contents

8.1 Introduction . . . 127
8.2 Component-Based Software Development . . . 129
8.3 User Interface Descriptions and Components . . . 130
8.3.1 The SEESCOA Component Framework . . . 130
8.3.2 The Rendering Component . . . 134
8.3.3 A Case Study: a Camera Surveillance system . . . 135
8.3.4 Decomposing tasks: relating components to tasks . . . 137
8.4 Discussion . . . 140

8.1 Introduction

In the previous chapters we focused on the task, dialog, and presentation models and we omitted the domain and application models. In this chapter we show how our approach can be integrated with a custom application model. The domain model will be presented as a *set of software components* here, although the approach is not limited to software components but is as applicable for webservices for example.

One of the results of the SEESCOA[1] [UBHB01] project is a common soft-

[1]Software Engineering for Embedded Systems using a Component-Oriented Approach, `http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/SEESCOA/`

ware platform, using components for embedded systems on a Java Virtual Machine. With this specific component-based approach for embedded systems, we can develop a framework for user interfaces adapting to the environment and device specific constraints as well as encourage reuse. The SEESCOA method is a component-based software development approach combined with ideas of contract-based specification for software objects. The SEESCOA components are considered as the application model here. These components provide the application logic within the interactive system.

Besides the application model, this chapter also focuses on *run-time migratable* user interfaces, which need to be independent of the target device, the target software platform and the interaction modalities. To enable migratable interfaces, a user interface can be considered as a presentation of a single service or several more functionally grouped services. These ideas are combined with a component-based approach allowing the designer to design user interfaces for particular components, which can be merged automatically at a later stage. This enables user interface designers to concentrate on what is important for multi-device user interfaces: how to present the user interface in a structured and logical manner. Notice this step makes the relation between the application models and the presentation model explicit.

Throughout this chapter we will use an example case study to illustrate the concepts we introduce: a small camera surveillance system using 4 different cameras. For each camera a software abstraction is provided by wrapping it with a SEESCOA component. Aggregating the four camera components by another component allows us to observe four cameras at the same time. Each camera has its own properties: some cameras can zoom in and out, other also allow to change the framerate,... All the components that are provided for this surveillance system make up the domain of this particular interactive system. Thus a domain is completely defined by the set of SEESCOA components that are used to implement the application functionality here.

This chapter is structured as follows: in section 8.2 a short introduction into Component-Based Software Development is provided. Since the focus of this thesis is not on Component-Based Software Development, we limit the discussion of this topic to the essential parts necessary to develop the remainder of this chapter. Continuing with section 8.3, we show how these descriptions can be combined with software components in general, and SEESCOA components in particular. The case-study is presented in more detail to show the results of the approach proposed here. This chapter also concludes with a discussion in section 8.4.

8.2 Component-Based Software Development

Central to Component-Based Software Development is the definition of a *software component*. One of the most generic definitions is introduced by Clemens Szyperski in [Szy98]:

> *A software component is a unit of composition with contextually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.*

Within the SEESCOA project, a component has a well defined description [com01]:

> *A component is a reusable documented entity that is used as a building block for software systems. It is used to perform a particular function in a specific application environment within a specific component system. Components are composed (glued together) using their interfaces. These interfaces consist of provided interfaces and required interfaces. A provided interface describes how the functionality has to be accessed. A required interface describes what is needed to perform this functionality.*

A clear distinction is made between the *component blueprint* and *component instance*. The blueprint is the description of the component, comparable with its interface and documentation. The instance is just the instantiation of this blueprint. Components are documented on four different levels:

1. the Syntactic level,
2. the Semantic level,
3. the Synchronization level and
4. the Quality-of-Service level.

SEESCOA components are targeted for embedded systems and can be reused easily in heterogeneous environments. Support for reuse is accomplished by adding Design-by-Contract[UBHB01] facilities to the component language. Based on the ROOM[2] notation, the SEESCOA notation provides ports serving as connectors between components. The SEESCOA component

[2]ROOM: Real-time Object-Oriented Modeling

system is completely asynchronous and uses the Java programming language as a common platform. Components communicate by sending asynchronous messages to each other, and not by using traditional synchronous message calls. For a full discussion of the SEESCOA methodology and run-time system, we refer to [UBHB01, UBB01]. Figure 8.1 shows part of the camera surveillance system case study (introduced in the next sections) in the SEESCOA component notation.

8.3 User Interface Descriptions and Components

8.3.1 The SEESCOA Component Framework

One of our involvements in the SEESCOA project is merging user interface design and component-based development for embedded systems. A traditional approach, making a "static" user interface as a layer on a service or a data layer has proven to lack flexibility. Based on the definitions given in section 8.2, we consider components as units that contain logically grouped functionality and data, each living in their own memory space. They should offer an abstract description of how the service or data offered by a (set of) component(s) can be presented. Think about components as software units offering a particular service through their interface: their interface is actually a description of their functionality. It is a natural extension to also allow components to describe what they want to offer to a human user. Each component can provide a description expressed in XML of the functionality it offers. Alternatively, they also could express in which way they could be interacted with. This is not true for all components of course (some just offer basic functionality on a lower level for other components), so only the components directly interested in human interaction should provide an abstract user interface description. This distinction allows us to define three different kinds of components from a interface design perspective here:

Internal components are components that implement functionality that will never be exposed directly in a user interface. These include the "core" components which encapsulate the basic functionality.

Surface components make up the layer that defines the functionality that has to be made explicit in the user interface and which can be used by the human user to interact with the system: this is the *application model* or *domain model* in Model-Based User Interface Development. Surface components can use or be composed out of internal components.

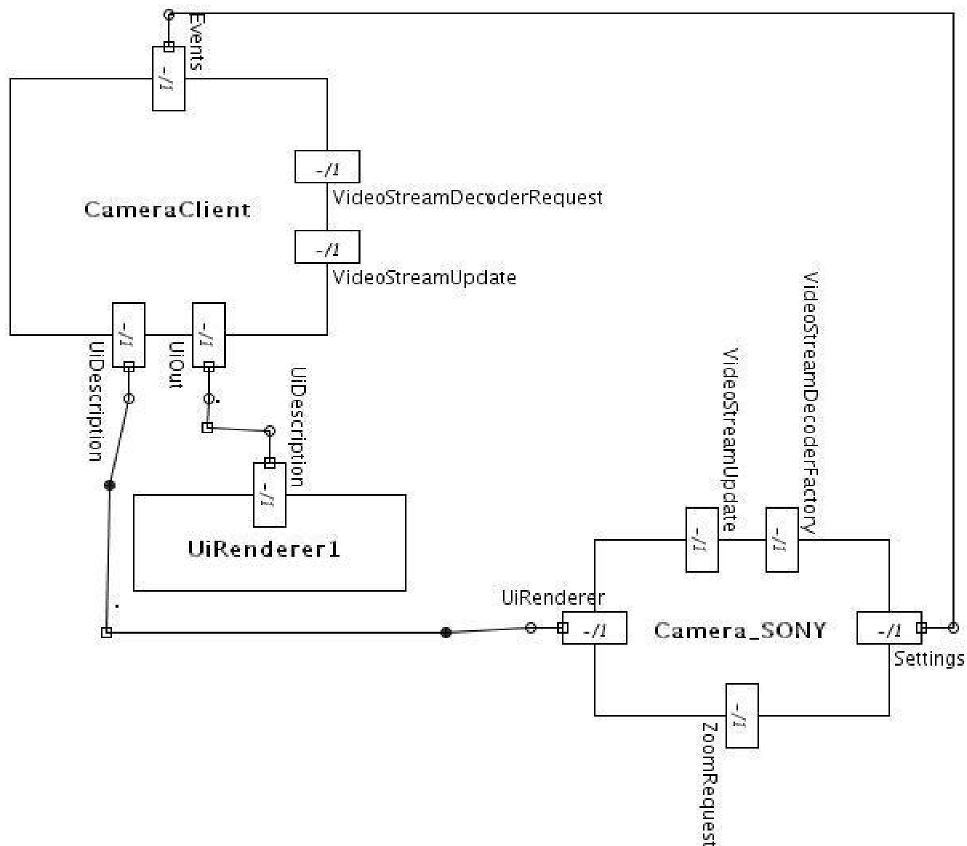


Figure 8.1: Part of the Camera Surveillance System in the SEESCOA component notation [RB03]

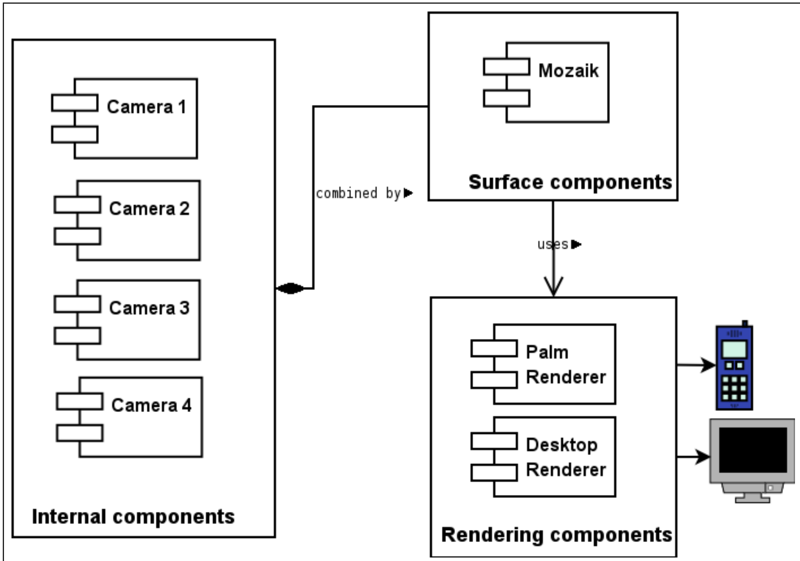


Figure 8.2: Surface, internal and rendering components in the Mosaic example

Rendering components are responsible to visualize the user interface. Rendering components have a predefined blueprint and communicate with the Input and Output devices on behalf of the surface components. Notice there can be several rendering components, thus supporting distributed user interfaces if the component system supports distributed communication.

The goal of this layered approach is to define the boundary between the application programmer and user interface designer. The interface designer only needs the (documentation of the) set of surface components, while the application programmer is mainly concerned with the internal components. The rendering components act as *services* for application programmers and interface designer and are not part of the application model.

When building applications out of components a user interface can be automatically derived from the set of surface components: each component provides its user interface in the form of an XML description [LVC02]. These XML descriptions can all be seen as subtrees of the final composed user interface description. I.e. the user interface will be automatically composed by connecting the user interface descriptions of the components into a complete user interface description. Figure 8.4 shows how this works using a small ex-

ample: the Camera Mosaic component which is described in more detail the next section (section 8.3.3). Each component can contain a description of their user interface: a description of a Camera can be found in listing 8.1 and of the Mosaic in listing 8.2. Figure 8.2 gives an overview of the components classified by the concepts introduced in the previous paragraph. Figure 8.4 presents how the descriptions can be combined at run-time to create the user interface out of the components. All components can export their user interface description, but only the surface components are responsible for the final user interface: if surface components use other internal components it is their responsibility to merge the different user interface descriptions accordingly and communicate with the rendering component(s) to handle user interface events.

Listing 8.1: user interface description of a single camera component

```
<group name="camera2">
  <interactor>
    <videowidget name="video">...</videowidget>
  </interactor>
  <interactor>
    <range name="zoomrange"><action>
      <func service="Surveillance.Controls">setFocus</func>
                <param name="camera2"/>
      <param name="zoomrange"/>
    </action></range>
  </interactor>
  <interactor><range name="focusrange">...</range></interactor>
  <interactor>
    <button name="camera1_onoff"><action>
          <func service="Surveillance.Controls">switch</func>
                <param name="camera2"/>
      <param name="camera1_onoff"/>
    </action></button>
  </interactor>
</group>
```

Listing 8.2: user interface description of a Mosaic component

```
<ui>
  <title>Camera mosaic</title>
    <group name="mosaic">
      <group name="camera1">&CAMERA1</group>
```

```
      <group name="camera2">&CAMERA2</group>
      <group name="camera3">&CAMERA3</group>
      <group name="camera4">&CAMERA4</group>
    </group>
</ui>
```

Notice this approach allows components to migrate and offer their services in other places. The user interface will integrate smoothly with other components on the target system: it can be connected to a surface component that will merge the user interface description with the descriptions of other components it aggregates, or can act as a surface component. The Component-Based Software Development approach supports a distributed view on assembling applications out of components and generating their user interface: parts of the user interface are allowed to migrate together with the functionality the components offer. Finally, the user interface description can be submitted to a rendering component as an XML document containing the user interface description. An example of such an XML document is given in listing 8.1.

8.3.2 The Rendering Component

As we take a Component-Based Software Development approach for designing user interfaces for embedded systems, there is one "basic" type of component that supports user interaction: the user interface rendering component. This component can be compared to a web-browser: a description for an interface can be submitted to the component and it will take care of rendering this description. Nevertheless, there are some differences: the component can receive a description of a user interface and render it to different kinds of output devices and widget sets. The state of the user interface can be "serialized" back into XML and relocated, which makes the component approach suitable for distributed systems or remote user interfaces. The SEESCOA component system takes care of the communication and makes it network transparent. Notice the rendering engine is also embedded in a component, so this component can also have a user interface description of its own functionality. To show its user interface the rendering component can send its user interface description to itself: it can be "bootstrapped".

Migratability of interfaces is implicitly supported because the SEESCOA component system is a distributed system. The rendering components can reside everywhere it has access to input/output hardware and can take input from every component no matter what its location is in the networked system.

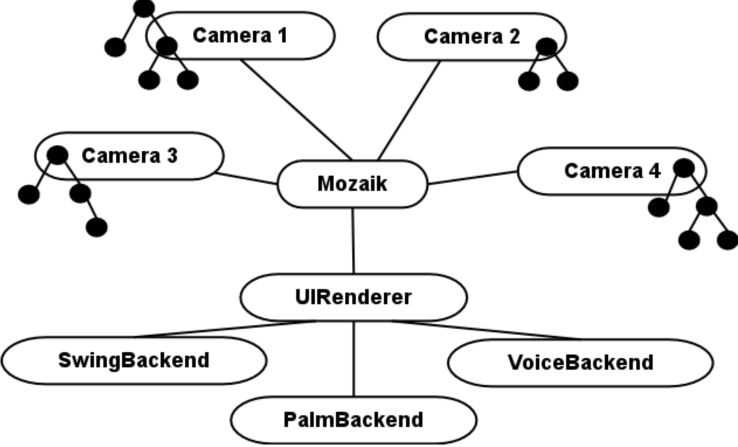


Figure 8.3: Component composition example: a simple camera surveillance system

This results in migratable user interfaces, provided the component infrastructure supports location independent communication between components.

8.3.3 A Case Study: a Camera Surveillance system

To illustrate how components can deliver their own user interface description, we developed an example case study in the context of the SEESCOA project: a surveillance system. The example surveillance system consists of 4 cameras, each camera is represented by a component. The system also contains a Mosaic component, combining the controls for each camera in a combined control (figure 8.1). The Mosaic component communicates with a rendering component which renders a user interface to an output device. The setup is presented in figure 8.3. Notice each camera component has its own user interface description (such as the one shown in listing 6.2) presented as an XML document. This is shown by the trees attached to the camera components in figure 8.3. Each camera may offer different possibilities so they can all have different user interface descriptions (The camera component is a component which abstracts the hardware and presents a physical surveillance camera).

Because of the possibility to specify hierarchical groups, the Mosaic component can take the four individual controls and add them as subtrees in a new tree. The Mosaic component only needs to add a new root with 4 groups as the children of the root node. Each control can be attached to a group node

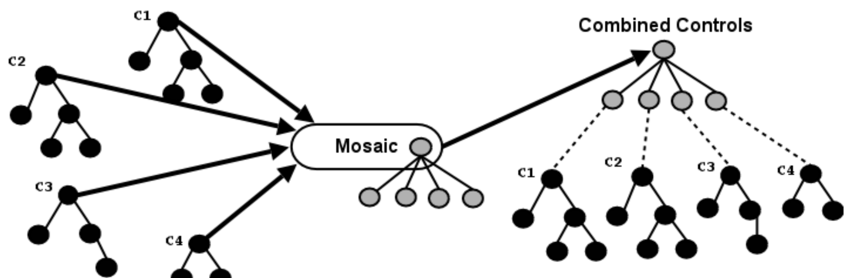


Figure 8.4: The Mosaic component combining several user interface descriptions

(figure 8.4, the group nodes are colored gray). The user interface description produced by the Mosaic component is passed to the rendering component and rendered according to the chosen back-end. This illustrates how combining components to access their provided functionality in one application automatically results in a combined user interface of these components. Notice several hierarchies can be mixed if desired: a subtree can be attached to an "open" node on another level in a new tree. This should be done with care: the chances of illogical and unusable generated user interfaces can increase by doing this. Our current system does not link the several subtrees across hierarchies, so no further support for mixing hierarchies is provided.

Depending on the target device the user interface for the Mosaic component will be different. Suppose for example we want to access the Mosaic component using a traditional desktop computer: the rendering component for a desktop PC will load the available CIOs and try to map the AIOs, [VB93]) described in the Mosaic user interface description on a widget set suitable for a desktop machine: figure 8.5 shows this. If we want to access the functionality of the Mosaic component using our PDA, the rendering component for a PDA will act the same: it will try to load the available CIOs and map the AIOs on this set of CIOs. This time the rendering component knows the PDA has limited possibilities, so it adapts the concrete user interface to the screen space constraints. Figure 8.6 shows the results using a PDA (Palm IIIc). The focus of this work was not data communication but run-time user interface migration, so we did not spend time investigating effective data communication between devices. The videostream for the PDA was actually implemented by sending separate down-scaled images to the device over its infrared connection. Of course, this can be done much more effectively using other techniques or

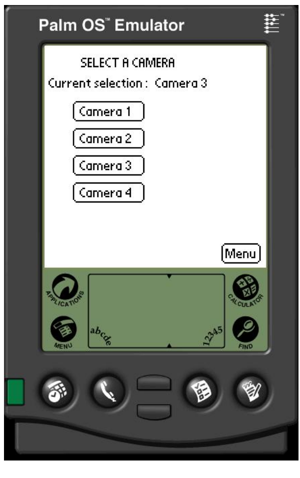
Figure 8.5: The Mosaic component on a desktop

means of communication.

8.3.4 Decomposing tasks: relating components to tasks

The case study introduced in section 8.3.3 is a very simple *interaction session* with a single dialog. We consider an interaction session as the interaction which happens to complete a subtask, like "select camera" in figure 8.7 for example. An interaction session can be represented by one or more user interface building blocks in a presentation unit. Most user interfaces have more than one interaction session: in a dialog-based user interface several dialogs are presented after each other according to the actions the user executes. A design method to take this into account is required at this stage. The design method should enable the designer to decompose tasks hierarchically, and link several interaction sessions to each other in order to achieve the postulated goal. This method should support a device-independent specification of the user interface.

To solve this problem, we combine our approach introduced in chapter 4 with our component-based description method. One of the advantages of the



(a) Pick a camera...

(b) ... and observe it

Figure 8.6: The Mosaic component on a PDA

ConcurTaskTrees notation is that we can extend it to model context-sensitive user tasks as described in [PLV01]. Characteristics that determine the context of use include the computing platform, the available interaction devices, available screen space,... When one or more of these characteristics change, a reconfiguration of the user interface may be required to adapt to the new context of use. [PLV01] proposes a notation to model context-sensitive user tasks. Their solution consists of a ConcurTaskTrees task model with roughly the following parts: a non-context-sensitive ConcurTaskTrees part and context-sensitive parts depending on some conditions. In [CLC04c] we extended this work with support for run-time context detection and multiple device interfaces. A more thorough explanation can be found in chapter 10. The second advantage is the asynchronous nature of the SEESCOA component system: ConcurTaskTrees allows to describe temporal relations, and includes concurrent tasks in its notations. A third advantage is the hierarchical structure it offers: our approach also uses a hierarchical notation to describe the user interface in a device independent manner.

Now suppose a human guard has access to a security system using a regular workstation or a PDA. Some tasks he can perform on the workstation are not possible on the PDA. Suppose for example that it is not possible to observe

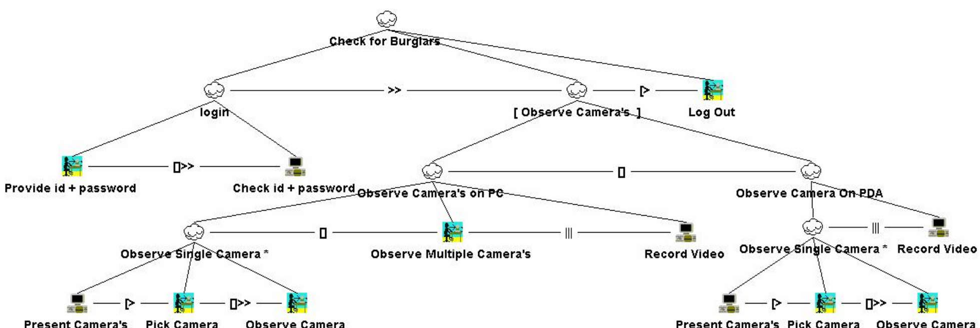


Figure 8.7: ConcurTaskTree diagram: checking for burglars with the camera surveillance system in a context-sensitive way

more than one camera at the same time on the PDA due to the minor screen space provided by it. So it depends on the context of use (the device that's being used in this case) whether the operator can pick just one or multiple cameras to observe at a time. Obviously we can say that this is a context-sensitive task. There are also a couple of non-context-sensitive tasks in this case. The operator must login to the system before he can pick cameras. Also he can choose to stop observing or pick other cameras to observe. While the guard is observing a camera (or cameras depending on the context) the other cameras will continue to record their video streams until the guard logs out again. The enhanced ConcurTaskTrees tree is shown in figure 8.7.

While being a good solution for modeling context-sensitive tasks there are two minor drawbacks to it. The first one is that some subtrees may appear more than once in the model. For example in figure 8.7 the subtree *Observe Single Camera* appears in the two different contexts of use. [PLV01] solves this by factoring out these subtrees by placing them in the context-insensitive part of the model. The second drawback is that we still have to model every possible context of use: for each device different properties have to be taken into account. In our approach we try to avoid this by using abstract user interface descriptions for an interaction task. A ConcurTaskTrees description can be saved as an XML document, which allows us to attach our own XML description at the leafs representing an interaction task. These XML descriptions are actually the composed descriptions of the components which are used at that moment. *A ConcurTaskTrees description becomes a way to describe how we want to interact with a set of given components in a particular stage of the usage of an application.* We gain a model-based approach for designing

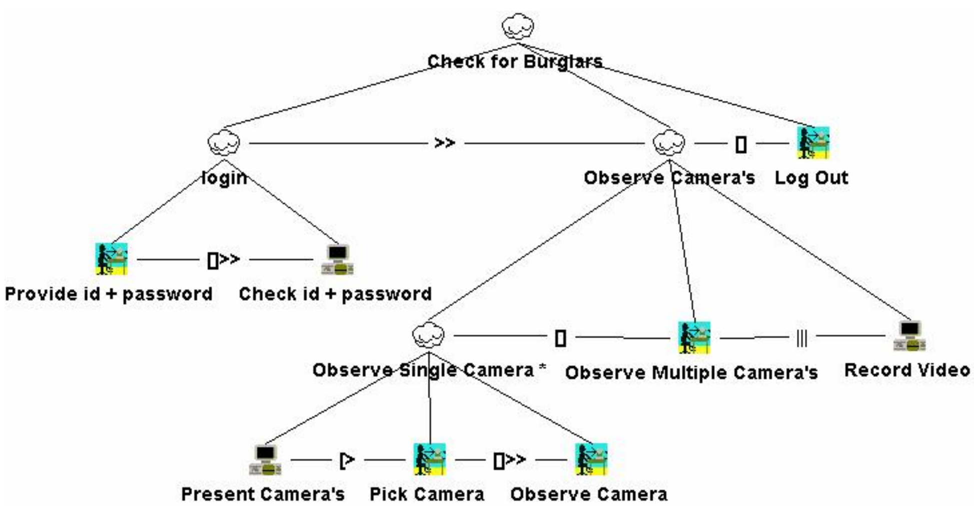


Figure 8.8: ConcurTaskTree diagram: checking for burglars with the camera surveillance system

the user interface, extending the component-based approach for modeling the software itself. So, instead of using a context-sensitive description as shown in figure 8.7 we can accomplish the same thing with a non context-sensitive description as shown in figure 8.8. We recognize that these are just the first steps, and the method has not been tested for a wider range of devices yet. When using totally different ways of interaction (e.g. not dialog-based), we expect we need context-sensitive parts as a consequence of particular other ways to complete the subtasks.

8.4 Discussion

In this chapter we presented a case study where Dygimes, and in particular the UiBuilder rendering engine, were successfully used for embedded systems. A camera surveillance system was implemented using solely the SEESCOA component system and its supporting tools, and the UiBuilder rendering engine for creating the user interfaces. The full case study architecture and implementation is discussed in [RB03]. The SEESCOA Component system served as the application (and domain) model in this case.

The integration with the SEESCOA components proves SEESCOA XML

can be used with a non-traditional application model. E.g. most application models rely on object-oriented concepts (classes and interfaces) being available, while SEESCOA XML provides an extensible action mechanism to use a multitude of application model types (web services, object oriented libraries, components,...).

This chapter also shows some Dygimes tools can be used separately. In this case UiBuilder is embedded as a component and acts just like the other components in the component system. It takes SEESCOA XML descriptions as input and generates an interface on the platform it resides on. Migratability of interfaces is implicitly supported because the SEESCOA component system is a distributed system: UiBuilder components can reside everywhere it has access to input/output hardware and can take input from every component no matter what its location is in the networked system.

A complete user interface is the aggregation of all components that have a SEESCOA XML description to specify the properties they want to offer to a human user. The system can generate its user interface by retrieving all components that are annotated with a user interface description from the complete set of components and merge these descriptions. The UiBuilder component will automatically generate the appropriate interface for the whole application w.r.t. the host platform.

Chapter 9

Uiml.net: an Open Uiml Renderer for the .Net Framework

Contents

9.1 Introduction **143**
9.2 UIML Overview **145**
9.3 Related Work **148**
9.4 The Renderer **149**
9.4.1 Overall Design 149
9.4.2 Dynamic Core 151
9.5 Inter-vocabulary distances **153**
9.6 The Layout Problem **157**
9.7 UIML and Dygimes **159**
9.7.1 Integration with the task specification 161
9.7.2 Generation of the dialog model 161
9.8 Discussion **164**

9.1 Introduction

The goal of this chapter is twofold:

- to verify the applicability of the Dygimes process with another presentation model,
- to evaluate the Java-based SEESCOA XML renderer w.r.t. other technologies.

Based on the findings of chapter 3 we select UIML as a worthy alternative for SEESCOA XML . Since UIML is above all useful as a notation for the *presentation model*, this makes it a good alternative for SEESCOA XML.

An important drawback in the framework introduced in the previous chapters is its inability to dynamically extend the AIOs and CIOs it is able to handle. Most High-Level User Interface Description Languages have this limitation because they attempt to define a general set of AIOs which are sufficient to built the most common interfaces. For example, in chapter 3 we showed Teresa XML, XForms, AUIML and UsiXML have a predefined set of AIOs. Unfortunately, once the design reaches the presentation level it remains difficult to specify this in a device-independent manner.

Chapter 3 pointed out there are very few High-Level User Interface Description Languages that succeed in being generic enough to be really independent of the widget set (e.g. some can only be used with Java widgets, or are only suitable with webbrowser support). The User Interface Markup Language (UIML) [APB$^+$99, Pha00] is a specification that is independent of a widget set and claims to be device-independent as well. Because the specification has matured over the years and efforts are emerging to submit it as a World Wide Web Consortium (W3C) specification, it is beneficial to develop renderers for the specification. Some of the current research efforts include creating better support for multi-platform user interfaces [FPQAS02, Pha00] and integration in Model-Based User Interface Development (TIDE, [FPQAS02]).

Targeting multi-device environments implies the UIML renderer has to be very flexible: on different devices there could be different widget sets, or the widget set API can be slightly different due to the different device profiles. This work also targets to create a UIML renderer that can manage and support evolution in widget set APIs and differences in widget set vocabularies without the need for changing the renderer itself. The renderer we implemented is designed to easily support a wide range of widget sets with a minimal effort. This is proved by the three widget sets it supports: *Gtk#*[1], *System.Windows.Forms* (SWF)[2] and *Wx.NET*[3].

The remainder of this chapter is structured as follows: section 9.2 gives a short introduction into the UIML language. It provides the necessary details of the specification to understand the following sections. Next, in section 9.3 some related work and underlying technologies are discussed evaluating

[1] `http://gtk-sharp.sourceforge.net/`

[2] `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemWindowsForms.asp`

[3] `http://wxnet.sourceforge.net/`

the use of UIML to illustrate the context of the work. This is followed by a description of the implemented renderer in section 9.4. Several aspects will be highlighted with the emphasis on the flexibility of the renderer. Section 9.6 identifies the layout management problem and proposes a solution for future High-Level User Interface Description Language renderers. In section 9.7 we illustrate the modular design of the Dygimes process by replacing SEESCOA XML for this purpose by UIML. This chapter is concluded with a discussion in the last section.

9.2 UIML Overview

The UIML specification is currently under revision for submission as a W3C standard. Consequently this means some changes in the specification can be expected and current renderer software design should support easy refactoring to adopt these changes.

An UIML document exists of several parts [AH04b] that are shown in figure 9.1. Together they form the Meta-Interface Model (MIM):

Interface describes four parts of the user interface:

- **Structure** : describes the "hierarchy" of the user interface. It defines the different parts that are contained in the user interface, and the interactor name of each part.
- **Style** : describes properties of the parts defined in the structure. This allows to change properties of the interactors like text, color,... The layout is also defined as a style of the parts in structure. Unfortunately the current way of defining a layout is not suitable for multi-device user interfaces, section 4.6 will elaborate further on this.
- **Content** : separates the content of the interface (e.g. the list of items that has to appear in a list presentation) of the other parts.
- **Behavior** : defines rules with actions that are triggered when some condition is met. Some kind of event mechanism is offered to the user interface designer this way.

Vocabularies are referred to as *peers* in the UIML specification: this contains the mapping with the concrete user interface toolkit. To allow the use of different devices and different GUI libraries, one can define several peers for the same UIML document while choosing the appropriate peer at

run-time. The renderer described in this chapter is limited to 2D widget sets.

Logic defines how to bind the user interface with the application logic. More precise it describes the mappings with the software interface to communicate with the application logic.

Listing 9.1 shows an example of a UIML document, where the different parts can be distinguished. Rendering the UIML document from listing 9.1 with the Gtk# vocabulary would result in the interface shown in figure 9.2.

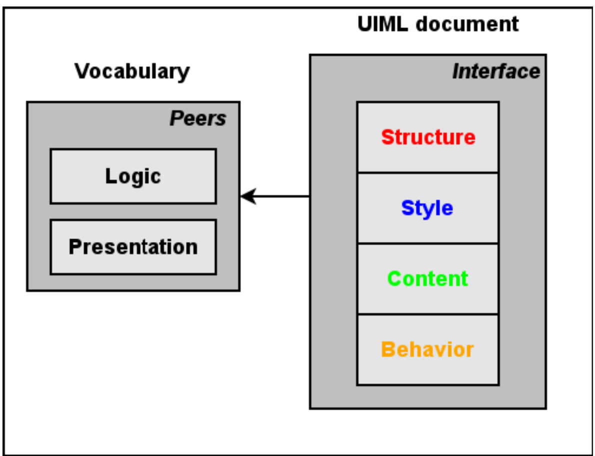


Figure 9.1: The UIML Meta-Interface Model

Listing 9.1: The UIML code for the Dictionary example depicted in 9.2.

```
<?xml version="1.0"?>
<!DOCTYPE uiml
   PUBLIC "-//Harmonia//DTD␣UIML␣3.0a␣Draft//EN"
                 "http://uiml.org/dtds/UIML3_0a.dtd">
<uiml>
  <interface>
      <structure>
    <part class="Frame" id="OuterFrame">
      <part class="HBox" id="hl1">
        <part class="VBox" id="vl1">
```

```
        <part class="Label" id="TermLabel" />
        <part class="Combo" id="TermList" />
      </part>
      <part class="VBox" id="vl2">
        <part class="Label" id="DefnLabel" />
        <part class="Text" id="DefnArea" />
      </part>
    </part>
  </part>
</structure>
<style>
  <property part-name="OuterFrame" name="label">Simple Dictionary
  </property>
  <property part-name="TermLabel" name="text">Pick a term:</property>
    <property part-name="DefnLabel" name="text">Definition:</property>
    <property part-name="TermList" name="content">
    <constant model="list">
      <constant id="Cat" value="Cat" />
      <constant id="Dog" value="Dog" />
      <constant id="Mouse" value="Mouse" />
    </constant>
  </property>
</style>
<behavior>
  <rule>
    <condition>
      <event part-name="TermList" class="EntrySelect"/>
    </condition>
    <action>
      <property part-name="DefnArea" name="text">
      <call name="Dict.lookup">
      <param>
        <property part-name="TermList" name="entry"/>
      </param>
       </call>
      </property>
    </action>
  </rule>
</behavior>
```

```
  </interface>
  <peers>
    <presentation base="gtk-sharp-1.0.uiml"/>
  <logic id="dictionary">
    <d-component id="Dict" maps-to="Dictionary">
      <d-method id="lookup" return-type="string" maps-to="Lookup">
          <d-param id="animal" type="System.String"/>
        </d-method>
    </d-component>
  </logic>
  </peers>
</uiml>
```

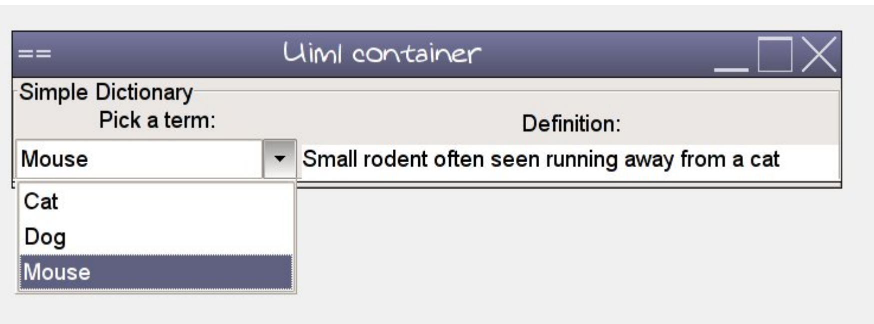


Figure 9.2: The dictionary example from listing 9.1 rendered with the Gtk# vocabulary.

9.3 Related Work

Until now, we are not aware of any previous work describing the actual implementation of an UIML renderer and releasing the source code. There were some initiatives in the past, but most of these projects only implemented parts of an obsolete specification version or are no longer supported. [BS02] describes how UIML can be converted into program code. Harmonia [FPQAS02] offers a Java-based UIML renderer that implements most of the specification. Several other implementations are gathered on `http://www.uiml.org`, but most of them are deprecated.

However, there are several research initiatives that are based on the UIML specification. DISL is a language developed by Bleul et. al. [BMS04] that extends UIML with a dialog model. Zuehlke et. al. introduced useML [ZMBR04]: a model-based, task-oriented and platform independent XML-based description language. By combining UIML and useML, UIML is enhanced with support for abstract models: it becomes independent of the platform and a task-oriented user interface structure is added. CUIML is an extension of UIML by Sandor and Reicher [SR01] that adds support for multimodal user interfaces (form-based, 3D and voice interfaces) and a controller in the sense of the Arch architecture [Cou93].

It is clear that UIML was designed with Object-Oriented programming languages in mind. Most mappings on the user interface toolkit and the relations with the application logic rely on the existence of "classes" in the OO sense. The most mature implementation of the UIML renderer is the one provided by Harmonia, and is implemented in Java. However, the .Net framework offers some new possibilities to develop a UIML renderer. For example it supports on-the-fly executable code generation and better integration with web services. This is the first attempt to write a UIML renderer for the .Net Framework.

Portability to different platforms and availability are important issues, so the renderer was both implemented on the Microsoft .Net Framework and the Mono (`http://www.go-mono.com`) implementation of .Net. Both .Net Frameworks implement the same ECMA standard, so the implementation was reusable as is for both .Net Frameworks. The UIML renderer we provide also works in the Compact .Net Framework and therefore is also available for PDAs and smart phones.

9.4 The Renderer

9.4.1 Overall Design

A High-Level User Interface Description Language can be processed in two different ways: either it can be *compiled* or *rendered*. The former transforms the specification into program code, the latter provides a rendering engine that can interpret the UIML document. When the UIML document is transformed into source code ("compiled"), on its turn the resulting source code needs to be compiled. Transforming the UIML document into program code is advantageous when the code still needs to be manually changed afterward.

The rendering approach is more complex to implement, but is more flexible:

it allows fast prototyping because an UIML document can be tested directly, it can offer dynamic changes in the user interface and a transparent mechanism for connecting the rendered user interface with the application logic.

Several parts of the renderer can be distinguished:

Vocabulary Generator One of the most cumbersome and tedious tasks is to create a vocabulary for a particular widget set. When the vocabularies are manually edited this often results in different incompatible vocabularies and incomplete mappings. When the widget set API gets updated, often the vocabulary has to be updated manually if one wants to support the latest version of the widget set. This process can be automated when the implementation language supports reflection, e.g. Java and C# have reflection support. Reflection allows software to inspect implementation code and APIs at run-time.

Interface reader In the initial stage the UIML document has to be processed. The Interface reader processes the document and stores it in an appropriate data structure. Notice that it is recommended to keep this data structure in memory during the lifespan of the user interface: dynamic changes in the style and the user interface structure can be supported better this way.

Style repository The style properties included in a UIML document are implemented in a repository-like manner. On the one hand the part that is specified beforehand is queried using XPath expressions. On the other hand there is support for properties that are added at run-time by an internal data structure.

Rendering Backends The specification allows different peers to co-exist for the same interface specification. A peer defines the language bindings for the interface, thus which widget set is being used and how it can interact with the application logic.

System Glue The system glue connects the concrete interface with the application logic. There are different ways to do this; by means of direct method invocation, remote method invocation or through web services. All three ways are supported by the .Net framework making it a powerful choice for implementing the renderer.

Figure 9.3 gives an overview of the architecture of the renderer. Uiml.net runs on the .Net Framework, which can use multiple widget sets. It takes

as input a UIML document, and reads the vocabulary which is referred from the UIML document. It loads the generic rendering core, and selects the appropriate rendering extensions for the widget set that is being used in the vocabulary. With this information it can render the user interface. Figure 9.4 illustrates the rendering process of the Uiml.net renderer. Uiml.net deserializes the UIML document into an internal data-structure (this is convenient for changing the user interface at run-time for examples). This structure is used by the generic rendering core which takes care of converting the part structure into a user interface hierarchy by using the mappings specified in the vocabulary. It also applies all the properties from the style fragment. The widget set specific rendering backend primarily adds support for special constructors and widget set specific properties that can not be handled in a generic way (e.g. a tree widget can be completely different among widget sets). The widget set specific rendering backend also has type converters that convert data from a string into a type that is suitable for the target widget.

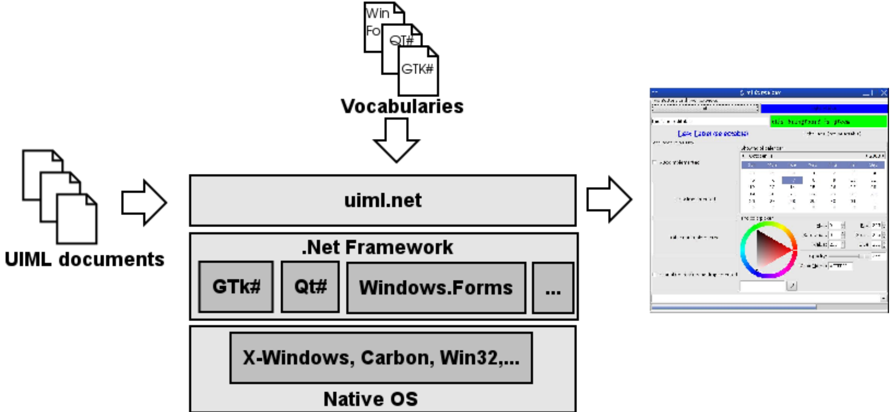


Figure 9.3: A rough sketch of the Uiml.net architecture

9.4.2 Dynamic Core

Roughly spoken there are two ways of implementing a renderer for a user interface markup language:

Static renderer The implementation relies on specific knowledge of the widget set. The types offered by the target GUI library are loaded and used at compile-time.

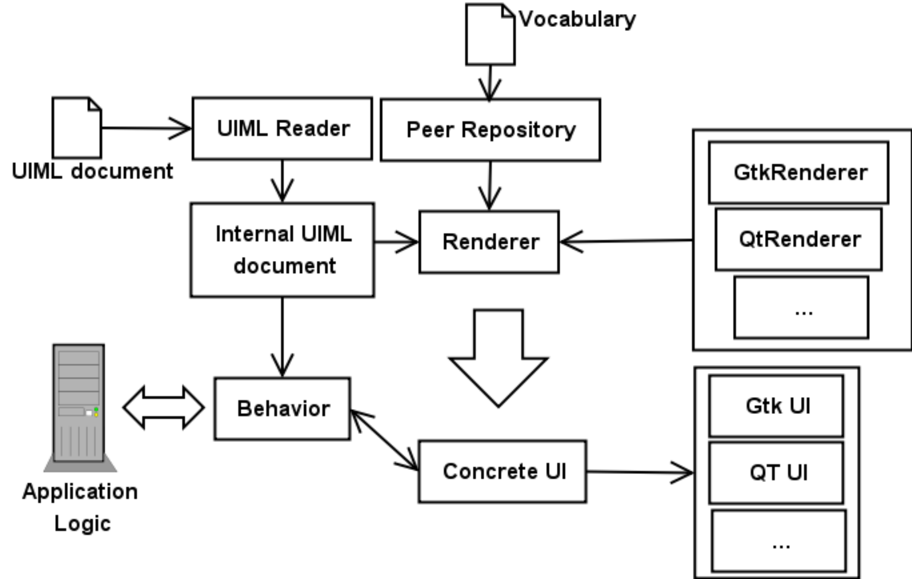


Figure 9.4: Processing an UIML file with Uiml.net

Dynamic renderer The implementation does not rely on specific knowledge of the widget set. The types offered by the target GUI library are loaded and used at run-time.

The former is more robust but less flexible and requires more program code. The latter takes full advantage of the information offered by the peer descriptions (vocabularies); it requires a detailed mapping description in the vocabulary however.

Reflection is a very powerful tool to use when mapping the AIOs onto CIOs [VB93]. AIOs are abstract representation of widgets, and CIOs are the concrete representation; e.g. a "range indicator" is an AIO and can be mapped to a slider widget (which is a CIO). In general, reflection is very powerful in the implementation of extensible XML-based User Interface Description Language renderers. The Uiml.net rendering engine itself has no notion of concrete widgets, but will be guided by the vocabulary to search for the appropriate mapping. Even when the concrete widgets are found (including its class name and properties), the renderer will avoid using the explicit class names. Instead it queries the available libraries containing possible widget sets with the information retrieved from the vocabulary. The reflection mech-

anism allows to construct new objects using solely this information. This has several advantages:

- The rendering engine is **reusable** for other widget sets, since it does not explicitly create the concrete widgets.
- The vocabulary can be extended **independent** of the renderer. When the widget set is updated, only the vocabulary has to be updated. New entries in the vocabulary can be used without further adaption of the program code.
- The renderer is more **portable** across devices. E.g.: it can be ported to platforms that only offer a limited version of the same widget set.

We tested the extensibility by adding a new widget set from scratch. Since we started with Gtk# and had a fairly complete rendering engine for this widget set, we added the SWF widget set afterward. This took one person (a 3rd year computer science student) without any previous knowledge of the code two days of work: a half day to understand the Gtk# specific rendering code, one day to create a SWF rendering backend and another half day to support event handling for this widget set. Most of the time was spent in creating the vocabulary for the SWF widget set, a task that could be partially automatized by a vocabulary generator. The only changes that had to be made were adding the necessary code to the SWF namespace; the original code did not change in any way to add this widget set! For each widget set that is supported there is a small piece of code that is written especially for this widget set however. This piece of code manages the “specificities” of the target widget set: widget constructors and the widget set containment hierarchy can be very different between different kinds of widget sets. Layout is another aspect that behaves very differently among different widget sets.

9.5 Inter-vocabulary distances

Since the implementation of the renderer itself is focused on being highly extensible, it is important to allow a designer to reuse a user interface design with another widget set while making minimal changes. The rendering backend that is responsible for creating the concrete user interface is selected by looking at the vocabulary the UIML document refers to. An optimal situation would be to only change the vocabulary reference to render the user interface to another widget set, but different widget sets have similar widgets in common that still have other properties.

Instead of identifying a generic vocabulary (a single vocabulary for different widget sets) like proposed in [FPQAS02], we follow another route to support multiple widget sets. With a generic vocabulary a trade-off has to be made: without an extra pre- or post-processing stage the specific capabilities of a widget set are limited by the generic vocabulary. In our approach the *inter-vocabulary distance* is minimized by following some simple rules when a vocabulary for a particular widget set is created:

- The vocabularies offer the same degree of abstraction in the mappings they define. There is a common set of interactors that appear in most vocabularies, each vocabulary that can be used with Uiml.net has support for Container, Button, Label, Frame, Entry, Text, Image, CheckButton, RadioButton, ToggleButton, ProgressBar, Combo, List, Tree, Calendar, ColorSelection, FontSelection, HorizontalRange and VerticalRange. These are all interactors that can be easily mapped onto widgets in a widget set. The vocabularies use the same naming scheme for the widget mappings: the semantics of a widget is used to determine its name in the vocabulary. All vocabularies use the same semantics, but could map this on different widgets.

- The layout of the parts from the structure fragment of a UIML document has to be generalized for the supported vocabularies, without the need to implement a complex algorithm inside the rendering engine.

- Event handling is independent of the widget set: the vocabulary should offer a widget set independent way for doing this. At the time of writing the most convenient way is to provide an observer for each widget as a callback function.

To show how this results in widget set independent building blocks that can be easily reused we provide a simple example of a user interface specified in UIML. We want to use both the Gtk# and SWF vocabularies to render the user interface and minimize the effort that is necessary to do this. Figure 9.6 shows the user interface rendered with the Gtk vocabulary and the SWF vocabulary. In figure 9.5 the differences between both UIML documents are shown graphically: left is the UIML document for the Gtk# example, and at the right the UIML document for the SWF example. Between the listings the differences are marked by solid filled quadrilaterals. Two important differences can be notice here:

1. The structure part of the user interface is the same *except for* layout-specific components

2. The properties from both documents share a general set of properties, and *differ in the specific properties* mostly induced by layout management specific properties.

For form-based user interfaces these two differences occur independent of the complexity of the user interface (e.g. user interfaces that contain much more widgets than the user interfaces from figure 9.6 are identical for the greater part except for the two differences detected here). The behavior section of the user interface is identical no matter which vocabulary is used. This leads to the conclusion that both the structure and properties part of a UIML document are insufficient to generalize the layout description of a user interface.

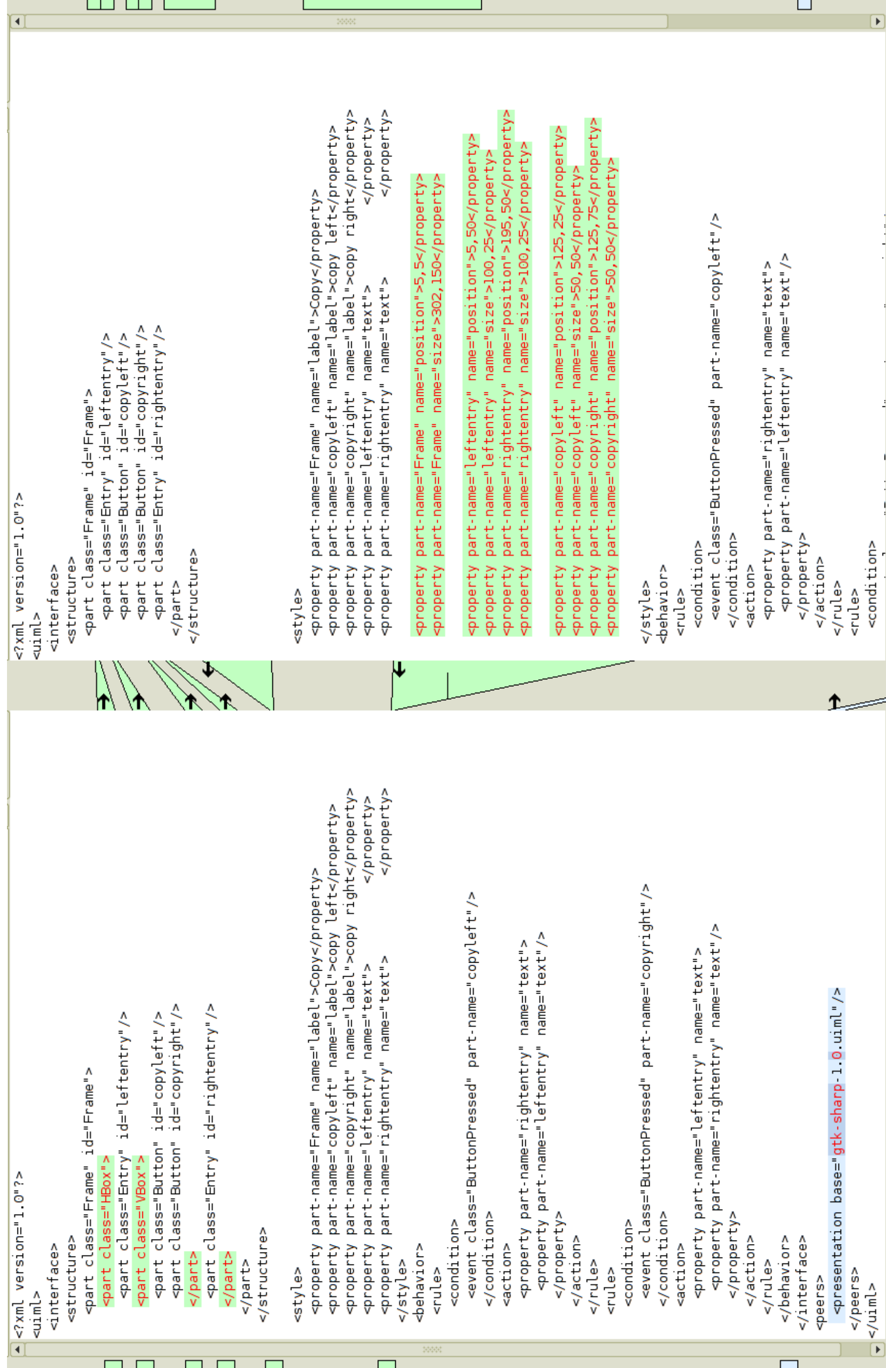
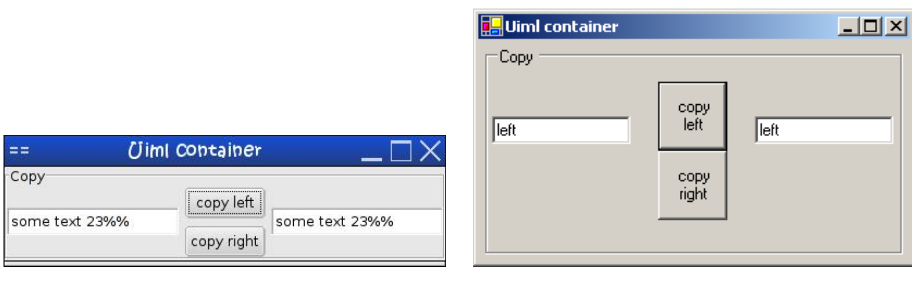


Figure 9.5: Differences between Gtk# and SWF interface shown by the Meld tool



(a) Gtk# vocabulary

(b) SWF vocabulary

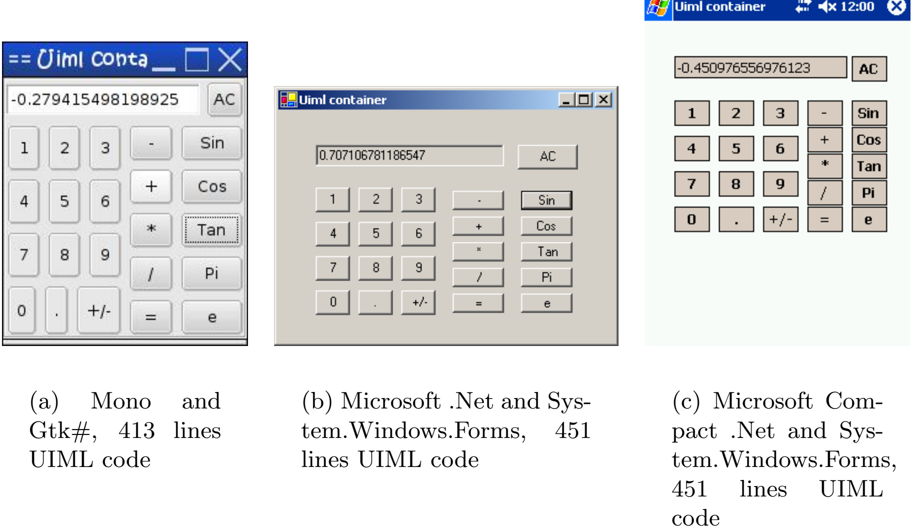
Figure 9.6: Copy Text example with UIML

9.6 The Layout Problem

As demonstrated in section 9.5, one of the pitfalls making UIML less suitable for multi-device interfaces is the lack of support for uniform layout management. As an example a calculator that is rendered on three different platforms is shown in figure 9.7. The Gtk# example differs with both System.Windows.Forms examples in 69 lines of code: the Gtk# example has a more elaborate part structure where containers are specified that handle part of the layout. This adds 12 part elements more in the structure for Gtk#, while there are 57 properties more in the System.Windows.Forms examples than there are in the style properties for the Gtk# example. Both the .Net and Compact .Net have the same amount of part tags and property tags, they still differ in the 57 properties there were necessary to specify a System.Windows.Forms layout. This is due the absolute coordinates that have to be used for that widget set. Nevertheless all three examples have exactly the same positioning of elements with respect to each other.

We propose to use spatial hierarchical layout constraints to overcome this problem. [MHP00] rightfully argues that constraint-based systems have not caught on for user interfaces, nevertheless *simple* constraints can be successful for specifying ("constraining") the layout of a system. Chapter 7 showed how a simple constraint-based layout manager can result in a very scalable (or "plastic") user interface.

Typically the layout of the user interface is influenced by the interface and style parts of the UIML document. Our approach differs with traditional



(a) Mono and Gtk#, 413 lines UIML code

(b) Microsoft .Net and System.Windows.Forms, 451 lines UIML code

(c) Microsoft Compact .Net and System.Windows.Forms, 451 lines UIML code

Figure 9.7: A calculator on multiple platforms and with multiple widget sets

approaches in the sense that we also use the hierarchy as described by UIML in the structure element instead of directly referring to the individual parts. Most available vocabularies have the layout specified as parts of the properties that can be defined in the style section of a UIML document.

In the way we implemented the renderer, the structure part determines how the concrete user interface will be nested and the style part specifies the more widget set related possibilities using layout managers. Using spatial layout constraints this separation can be preserved, while adding adaptability when rendering the user interface. Constraints are only defined between siblings in the structure tree. 9.8 shows the kind of flexibilty that has is supported by SEESCOA XML, but is not included in Uiml.net: 9.8(a) shows a “desktop” interface for browsing pictures, and figure 9.8(b) shows a rearrangement so the part of the user interface for navigating through the pictures can be used on a PDA. The layout constraints for the five pictures that control the large picture is shown in figure 9.9. The hierarchy divides the interface in groups and these groups can be subdivided in other groups and so on. All widgets that are part of the same group, have a logical relation with respect to each other. Some rules can be applied here:

- A group describes a set of **logically related** abstract interactors or groups of abstract interactors. The designer should decide which widgets are gathered in a group. In UIML syntax this could be obtained by adding a property to a part that serves as a container: this property can specify the semantic relation between its children.
- A group can be specified **splittable** (as a UIML property for that part). This specifier allows the layout manager to show the abstract interactors or groups of abstract interactors in separate spaces.
- The group specifier **non-splittable** (as a UIML property) forces the layout manager to show the children of the group as a whole to make sense to the user. Note that "non-splittable" is only valid for the direct children of the group, and does not affect the further offspring.

The type of semantic relation between sub-parts of the same part is specified in the general vocabulary and can be used as hints for the layout manager. E.g. a pagination algorithm as the one RIML [KWWZ04] uses could be useful when migrating the interface to a device with less screen space: the "splittable" property is sufficient to support this kind of algorithms.

For now, we have not implemented this into the Uiml.net renderer because it makes the renderer less portable. The layout management should be generic and not related to any widget set and modalities. By consequence this requires adding new elements into the UIML specification, e.g. tags to define constraints. The spatial constraints are implemented in the Dygimes framework [CLV^{+}03, LCC03] for testing purposes and has proven to be a feasible solution for user interfaces containing a limited amount of widgets. Results obtained in this experiment to combine a High-Level User Interface Description Language with spatial constraints can be seen in figure 7.3. The figure shows a hotel registration form described in a High-Level User Interface Description Language that is rendered to different devices.

9.7 UIML and Dygimes

This section shows how UIML can be used as a replacement presentation specification in the Dygimes Framework and run-time environment. In our approach, we can provide UIML with a dialog model and task model without changing or extending the UIML language itself. While most other approaches try to integrate this kind of models in the UIML specification itself; we constraint UIML to the presentation and application model. More specific: we

(a) On a desktop

(b) The controls rendered separate

Figure 9.8: The Multi(ple)-device Picture Browser with UIML

constrain UIML to the description of isolated dialogs. Looking at the different parts of a UIML document it is clear *structure* and *style* only apply to the user interface itself. The *behavior* part describes only the behavior *inside* a dialog in our approach. In the *peers* part, the *logic* section defines the application model.

For each dialog in the user interface, we need a complete UIML description to create the final user interface for this dialog. The final UIML description is obtained by merging different parts of a UIML description that are related with the different models. This can be expressed with the concepts introduced in chapter 2. Using another markup language with the Dygimes process is just a matter of splitting the new markup language in the appropriate sub-parts, and relate these sub-parts to the different models that are being used. These sub-parts were introduced in section 6.1: structure, layout, rendering hints and mappings. In the next subsection we will show it is sufficient to relate structure sub-parts as user interface building blocks to tasks.

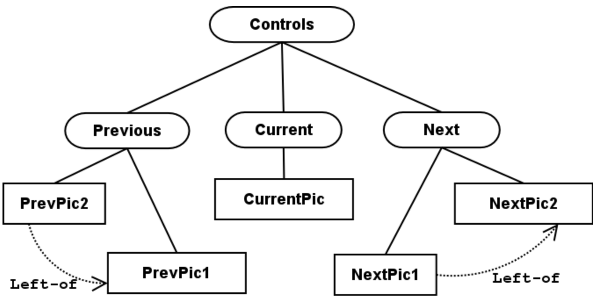


Figure 9.9: Layout constraints for the controls of figure 9.8

9.7.1 Integration with the task specification

A leaf task from an hierarchical task description is visualized by a *user interface building block*. This user interface building block contains an abstract description of a part of a user interface that can be considered as a “unit”: it would not be meaningful to split up the user interface any further. Figure 4.4 shows how this was done with SEESCOA XML (since this is a prototype tool, the user interface building block could also be presented graphically by the tool instead of as the XML listing). If we would like to use UIML it is only a matter of replacing the SEESCOA XML descriptions with UIML structure sub-parts.

Let us reconsider the simple task specification in figure 5.9(a). We could create user interface building blocks for each leaf task that should be visualized. This gives us the set of user interface building blocks shown in table 9.1. For each task there is a UIML document that visualizes the task. For clarity, only the structure fragment of the various building blocks is shown in table 9.1.

9.7.2 Generation of the dialog model

The set of dialogs necessary to have full coverage are generated automatically from the task specification: this is accomplished by the algorithms from chapter 5. Since a dialog is actually a mapping from a set of leaf tasks onto a presentation unit, the user interface building blocks attached to these tasks make up the presentation unit.

To know which dialogs are related to the task specification, we use the algorithms of chapter 5. This gives us the following enabled task sets for the

Task	*UIML structure*	*UI building block*
Enter Name	<part class="VBox"> <part class="HBox"> <part class="Label" id="lsurname"/> <part class="Entry" id="surname"/> </part> <part class="HBox"> <part class="Label" id="lname"/> <part class="Entry" id="name"/> </part> <part class="Button" id="ok"/> </part>	Uiml container Surname Name OK
Select Male	<part class="HBox"> <part class="CheckButton" id="male"/> </part>	Uiml contai Male
Select Female	<part class="HBox"> <part class="CheckButton" id="female"/> </part>	Uiml co Female
Choose Current Job	<part id="Top" class="Frame"> <part class="VBox"> <part class="CheckButton" id="lcurrent"/> <part id="currentjob" class="List"/> </part> </part>	Uiml co Choose current Job Employed Job Types Manager Scientist Salesman President Pilot Nurse
Select Unem-ployed	<part class="HBox"> <part class="CheckButton" id="unemployed"/> </part>	Uiml contain Unemployed
Submit	<part id="stop" class="Frame"> <part id="submot" class="Button"/> </part>	Uiml container Is all the Information correct? Submit

Table 9.1: User Interface building blocks for each leaf task from figure 5.9(a)

task specification:

$$
\begin{aligned}
ETS_1 &= \{Enter\ Name\} \\
ETS_2 &= \{Select\ Male, Select\ Female\} \\
ETS_3 &= \{Choose\ Current\ Job, Select\ Unemployed\} \\
ETS_4 &= \{Submit\}
\end{aligned}
$$

So we have to provide four different dialogs to have a full dialog coverage for this task specification. Take ETS_3 for example: it has two tasks, which should be visualized in the same period of time. Merging the user interface building blocks for these tasks is easy: it is just a matter of creating a new "container" part and adding the two structure fragments as children of this container in a new UIML document. The container part can be structured in several ways, depending on the designer preferences. Table 9.2 shows two possible containers: the left container just uses a horizontal layout and the container on the right uses tabbed pages. It is a trivial task to automate the merge of the UIML building blocks in these containers. In the example the UIML building block of the task "Choose Current Job" is inserted first, and the task "Select Unemployed" is inserted second in the container. The properties are not shown in the example because merging them is even simpler: the list of properties of the building blocks can be concatenated as can the behavior rules. Containers are simple layout patterns that could be pre-generated for the designer and customized afterward. In this example we made the completion of the enabled task set ETS_3 explicit in the container by adding a part "confirm" that maps on a button to indicate the information for this dialog is complete.

To make this work in Uiml.net, it is actually sufficient to support the template element. A template is described by the UIML specification as follows [AH04b]:

> UIML templates enable interface implementers to design parts of their UI (or even the entire UI itself) as reusable components to be exploited by other UIs. [...] Templates work as follows. Most elements can contain the source attribute; let such an element be E. The source attribute names a `<template>` element (either within the same document or in another document). The `<template>` element named must be an element of the same type as the element E (i.e., have the same tag name). The source attribute causes the body of the element inside the `<template>` to be combined with the body of E.

This is not yet supported in Uiml.net, but should pose no real problems since most of the necessary code is already available. The comment

```
<!--  insert UIML structure first task here  -->
```

could easily be replaced by

```
<part id="task1" source=#task1" how="replace"/>
```

where the UIML structure with identifier `task1` is a template that presents the user interface building block for task 1.

The last step to create the dialog model is the detection of transitions between the different enabled task sets. The dialog model is created in an identical way as the one previously presented in chapter 5. For this example the resulting dialog modal is shown in figure 5.9(b).

9.8 Discussion

We discussed the implementation of a User Interface Markup Language (UIML) renderer and possible extensions in this chapter like obtaining better adaptability for multiple/multi-device environments throughout the usage of spatial constraints. We explored how we could create a renderer that supports late-binding of a widget set with its application model through reflection, instead of binding with a widget set at compile-time. This results in a rendering engine that is extremely extensible and supports evolution of widget set vocabularies.

To demonstrate the reusability of the Dygimes approach with another XML-based User Interface Description Language, section 9.7 showed how UIML could easily replace SEESCOA XML and take advantage of the support for the task model, dialog model, and inter-model transformations that are available in Dygimes. There are also three important comparisons between SEESCOA XML and its UiBuilder rendering engine and UIML and its Uiml.net rendering engine possible here:

1. Java versus .Net for embedded systems and mobile devices
2. Java versus .Net for their user interface support
3. UIML versus SEESCOA XML

We will discuss briefly our conclusions for these three comparisons:

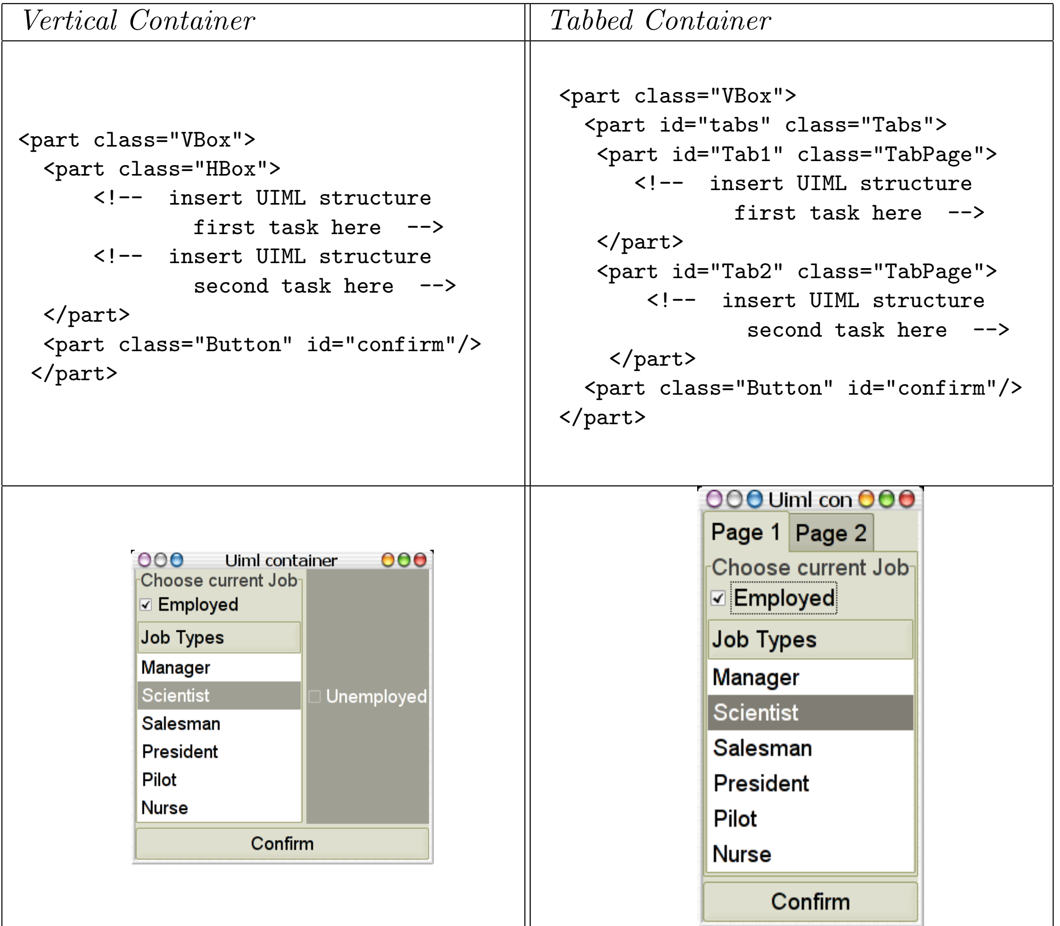
1. Java is still more suitable for developing software targeting embedded systems and mobile computing devices. Java is more widespread and supported on different operating systems. The Compact .Net Framework is only supported on the Microsoft operating system. Different profiles

allow Java to be "deeply" embedded. On the other hand, the Compact .Net Framework is not limited to one programming language.

2. The .Net framework has many potential candidates for implementing GUIs like bindings for XWindows, Gtk, Qt and the Eclipse-based SWT. However, Java has more suitable widget sets available that are scalable for a wide range of devices. The major drawback is there is less choice in "backend" widget sets because the .Net Framework allows smooth integration with native libraries while this is more difficult with Java.

3. UIML is a far more powerful User Interface Description Language than SEESCOA XML: it provides a generic way to describe the UI. It does not restrict the User Interface Description Language syntax to a particular set of widgets. The SEESCOA XML does restrict the available widget set in favour of better support for embedded systems. In contrast with SEESCOA XML, UIML is less suitable for embedded devices: it lacks an appropriate method for layout specification and needs a predefined GUI library vocabulary ("peers") to use a particular widget set.

The source code of Uiml.net can be obtained at `https://sourceforge.net/projects/uimldotnet/` and is licensed under the Lesser General Public License v2.1. Uiml.net supports Gtk#, System.Windows.Forms and part of the Wx.NET widget sets and works on the Microsoft .Net Framework, Mono and the Microsoft Compact .Net Framework (for PDAs and smart-phones).



Vertical Container	*Tabbed Container*

Vertical Container

```
<part class="VBox">
  <part class="HBox">
      <!--  insert UIML structure
              first task here  -->
      <!--  insert UIML structure
              second task here  -->
  </part>
  <part class="Button" id="confirm"/>
 </part>
```

Tabbed Container

```
<part class="VBox">
  <part id="tabs" class="Tabs">
   <part id="Tab1" class="TabPage">
      <!--  insert UIML structure
              first task here  -->
   </part>
   <part id="Tab2" class="TabPage">
       <!--  insert UIML structure
               second task here  -->
    </part>
  <part class="Button" id="confirm"/>
</part>
```


Table 9.2: Two examples of merging UIML building blocks from an enabled task set with a predefined container.

Part III

Towards Context-Sensitive Model-Based User Interface Development

Chapter 10

Extending Dygimes for Context-Sensitive User Interface Development

Contents

10.1 Introduction . . . 169
10.2 Related Work . . . 170
10.3 Dygimes Once Again . . . 171
10.4 Design Process . . . 172
10.4.1 The Context-Sensitive Task Model . . . 173
10.4.2 The Presentation Model . . . 175
10.5 A Case Study: Manage Stock . . . 177
10.6 Discussion . . . 183

10.1 Introduction

Recent advances in mobile computing devices and mobile communication support more complex interaction between different devices. This allows users to migrate from their single "computer on the desk" setup to a heterogeneous environment where she/he uses several devices to accomplish her/his tasks. Although the provided hardware and software becomes more powerful, it makes designing the interface more complex. Different contexts (device constraints, environment of the mobile user,...) have to be taken into account. The nomadic nature of future applications also demands a new way to design interaction using multiple devices. We use the following definition, based on

Dey's definition of a context-sensitive system [DSA01] and with the focus on sensing the environment in order to pursue uniformity and clarity in this work: *Context is the information gathered from the environment which can influence the tasks the user wants to, can or may perform.*

Combining the work presented in the previous chapters with context-sensitive task specifications like presented in [PLV01, SLV02], we can realize a supporting framework for the design and creation of context-sensitive multiple- and multi-device interaction. By multiple-device interaction we mean the user interface is distributed over different devices. The implementation has been tested as a component of the Dygimes framework [CLV+03].

The remainder of this chapter is structured as follows: section 10.2 discusses the related work, introducing the state of the art in context-sensitive task modeling. This is followed by an overview of the design process needed to create a context-sensitive user interface in section 10.4. In section 10.3 the Dygimes process is repeated but with a focus on context-sensitive user interface design. Three stages are described: the creation of the task model, the extraction of the dialog model and the generated presentation model. Next a case study shows how things work in practice. Finally, the obtained results and their applicability are discussed at the end of this chapter.

10.2 Related Work

Pribeanu et al. [PLV01] proposed several possible approaches to adapt the ConcurTaskTrees notation for ıcontext-sensitive task modeling. As pointed out in [PLV01] and [SLV02], the context of use of the application influences which parts of the task model are executed. A context-sensitive (or context-dependent) and a context-insensitive (or context-independent) part of the task model can be identified and processed accordingly. The context-sensitive part can be related to the context-insensitive part in multiple ways [PLV01]:

- Both parts are specified in one task model: the *monolithic approach*
- The context-insensitive parts are connected to the context-sensitive parts with general arcs: *graph-oriented approach*
- The context-insensitive parts are connected to the context-sensitive parts with special arcs that can constitute a decision tree: *separation approach*

The last approach in particular is interesting: although it allows different parts for different contexts of use to be integrated in one model, there is a *decision*

tree that provides a nice separation. We choose to insert *decision node* in the task specification instead of decision trees. Of course, decision nodes can have other decision nodes as descendants. The children of a decision node are possible subtrees where one of them will be chosen in a preprocessing step. Section 10.4 explains in detail how a concrete task specification can be obtained by preprocessing the decision nodes.

Paternò and Santoro [PS02] present a method to generate multiple interfaces for different contexts of use starting from one task model. In contrast with their approach, we do not focus on the design aspect as much as they do, but emphasize the run-time framework necessary for accomplishing this. To our knowledge, the ıTeresa tool supports the creation of one task model for multiple devices, but currently does not take into account multiple devices interacting at once or the interface migrating from one device to another.

Calvary et al. [CCT00, CCT01] describe a process where a *Platform* and *Environmental Model* are used to represent context information. The process allows to create user interfaces for two running systems in different contexts. Although at several stages in the user interface design process (Task Specification, Abstract user interface, Concrete user interface, Runtime Environment) a translation can take place between the two systems, the designer will have to change the task specification manually in the process if the context has an influence on the tasks that can be performed.

Nichols et al. [NMH+02] defined a specification language and communication protocol to automatically generate user interfaces for remotely controlled appliances. The language describes the functionalities of the target appliance and contains enough information to render the user interface. In this case, the context is secured by the target appliance represented by its definition.

Ali and Pérez-Quiñones [FPQAS02] also use a task model, together with ıUIML [APB+99], to generate user interfaces for multiple platforms. The task model has to increase the abstraction level of the UIML specification, which is necessary to guide the user interface onto different devices.

10.3 Dygimes Once Again

Most of the work presented in this chapter is built upon our framework ıDygimes (chapter 4). One of the aims in this chapter is to extend this framework to support design for context-sensitive user interfaces through selected models from Model-Based User Interface Development.

The Dygimes framework supports roughly the following steps for creating user interfaces:

1. Create a context-sensitive task specification with the ConcurTaskTrees notation
2. Create user interface building blocks for the separate tasks
3. Relate the user interface building blocks with the tasks in the task specification
4. Define the layout using constraints
5. Define custom properties for the user interface appearance (e.g. preferred colors, concrete interactors,. . .)
6. Generate a prototype and evaluate it (the dialog model and presentation model are calculated automatically)
7. Change the task specification and customizations until satisfied

This is a clear variation on the "traditional" multi-device user interface creation process that was presented in chapter 4. The basis of the process is step 1: the creation of a context-sensitive task specification.

10.4 Design Process

The proposed approach extends the process for automatically generating prototype user interfaces from annotated task models that include user interface building blocks. Figure 10.1 shows the extended process where a context-sensitive task model is considered to generate user interfaces depending on the context at the time the user interface is rendered. First, a ıcontext-sensitive task model is constructed and high-level user interface building blocks are attached to the leaves as described in the previous section. Next, the context is captured and the proper context-specific ConcurTaskTrees specification will be generated automatically. Subsequently the enabled task sets are calculated.

After this step, the appropriate dialog model is extracted automatically from the task model like shown in chapter 5. Each dialog is still related to the set of tasks it presents, thus also to the appropriate user interface building blocks it can use to present itself. The context-sensitive information in the task specification is taken care of in a "preprocessing" step, which we will explain now into further detail.

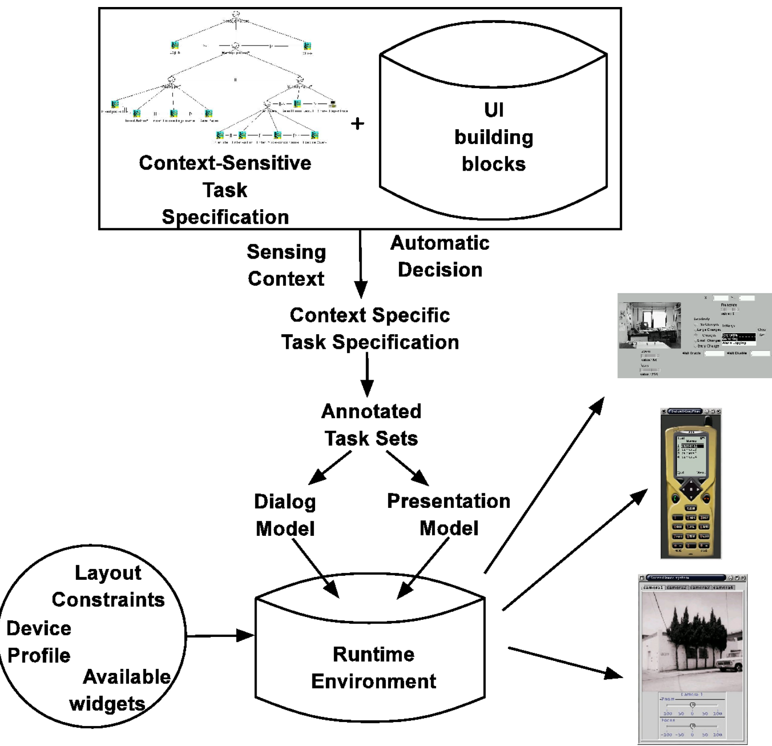


Figure 10.1: Context-Sensitive user interface Design Process

10.4.1 The Context-Sensitive Task Model

As pointed out in section 10.2, there are three proposed approaches to model context-sensitive task models. Our approach is inspired by the third one, the separation approach, but instead of collecting decision trees, we propose another way where the context-insensitive part points directly to context-sensitive subtrees through *decision nodes*. These nodes are marked by the **D** in the example in figure 10.2. Although this resembles the graph-oriented approach, the context-sensitive subtrees are the direct children of the decision node. When the context-sensitive parts are resolved, the decision node will be removed and replaced by the root of the selected subtrees of that decision node.

The decision nodes are executed in the first stage of the user interface generation process. This results in a normal ConcurTaskTrees specification, but also one that is suitable according to the rules defined in the decision nodes.

The normal ConcurTaskTrees specification enables the provided algorithm to extract the dialog model automatically adapted to the current context.

In order to link the context detection and the task model, some information about which subtree has to be performed in which case is added to the ıdecision node. Listing 10.1 shows a simple scheme (as a Document Type Definition) defining how rules can be specified for selecting a particular subtree according to a given context. Conditions can be defined recursively and numerical and logical operators are provided $(=, <, >, \vee, \wedge)$ to cope with several context parameters. In listing 10.2 an example is presented where the current context will be decided on the basis of comparing X and Y coordinates provided by a Global Positioning System (ıGPS) module. The XML specification provides a way to exchange context information. Tool support should be provided to hide the complexity of the decision XML for the designer.

Note the approaches described in [PLV01, SLV02] focus on the design of the interface at the task level. This work shows how the task model is used at run-time to generate context-dependent user interfaces. This is accomplished by providing a framework (Dygimes, section 4.2) that can interpret a task specification and generate a presentation for the given task specification. The framework resolves the context dependencies beforehand, resulting in a presentation that is adapted to the context of use. The next section explains how we proceed from the task specification to the presentation of the user interface by using a dialog model.

Listing 10.1: Decision DTD

```
<?xml version="1.0"?>
<!ELEMENT decision ((cond,true,false)
         |(value,case+))>
<!ELEMENT cond (value,value)>
<!ATTLIST cond type CDATA #IMPLIED>
<!ELEMENT value ( cond | #PCDATA )>
<!ATTLIST value type CDATA #IMPLIED>
<!ELEMENT true (#PCDATA)>
<!ATTLIST true platform #IMPLIED>
<!ELEMENT false (#PCDATA)>
<!ATTLIST false platform CDATA #IMPLIED>
<!ELEMENT case (value|cond)>
<!ATTLIST case platform CDATA #IMPLIED>
```

Listing 10.2: Decision XML example

```
<decision>
  <cond type="and">
    <value type="cond">
      <cond type="lt">
        <value type="context">
          GPS:Xcoord
        </value>
        <value type="int">
          1
        </value>
      </cond>
    </value>
    <value type="gt">
      <cond type="equals">
        <value type="context">
          GPS:Ycoord
        </value>
        <value type="int">
          54
        </value>
      </cond>
    </value>
 </cond>
 <true platform="context">left</true>
 <false platform="context">right</false>
</decision>
```

10.4.2 The Presentation Model

The last step of the adjusted design process has to render the dialog model on the available output devices. This is the presentation of (the different parts of) the concrete user interface. Because the decision nodes have their own rule set, the resulting task specification can use different devices to reach the user's goal. Several tasks from different subtrees can be executed in the same period of time, it is even possible the user interacts with *multiple* devices: the user interface for that set of tasks can be distributed over several devices [VC04]. The case study in the next section will show how different devices can be involved when executing the task specification.

The nodes in the dialog model are enabled task sets. One such node repre-

sents all user interface building blocks that have to be presented to complete the current enabled task set (section 4.2 showed that user interface building blocks were attached to individual tasks). The tasks in an enabled task set are also marked with their target device, so two different situations are possible:

1. *All* tasks in an enabled task set are targeted to the same device.

2. *Not all* tasks in an enabled task set are targeted to the same device.

Situation (1) allows the user interface to be rendered completely on one device. (2) demands that the user interface is *distributed* over different devices. For this purpose the device-independence of the abstract user interface description has to be extended towards the use of multiple devices. On the level of the presentation model, the high-level user interface description of a dialog are rendered as concrete dialogs, this can be accomplished by using two important techniques:

- Customized mappings from Abstract Interaction Objects (AIOs) to Concrete Interaction Objects (CIOs) [VB93]. The rendering engine for each device can choose for itself the concrete widget selected to present an AIO. This can be customized afterward by the designer [VLC03a].

- Positioning of the widgets is done through constraints which are defined in a language-independent manner. The renderer can use the information about the hierarchical widget containment to split up the user interface in different parts. Details of this approach can be found in [LCC03] and chapter 7.

Customized mapping rules and device-independent layout management are two important techniques for realizing device-independent distributed user interfaces.

It is possible several concurrent tasks located in the same enabled task sets have to be rendered on different devices. Since the presentation building blocks are attached to the tasks as XML documents, the presentation for an individual device can be calculated for each device separately. Notice when concurrent tasks are rendered on separate devices, some kind of middleware will be necessary to support data-exchange between both tasks in a heterogeneous environment. In contrast with e.g. WebSplitter [HPN00] the focus is not on distribution of content, but distributed support of task execution.

10.5 A Case Study: Manage Stock

Figure 10.2 shows the *manage stock* example. The following situation occurs: the storekeeper of a warehouse keeps track of the stock using two devices. First a desktop PC is used to manage the purchase and sales of articles. Second an employee checks and updates the stock amounts using his PDA to note the changes immediately. When the amount of a certain article is updated by the desktop PC (see figure 10.4), for example when new goods are purchased, the employee receives a message on his PDA. When he/she stands in the vicinity of a printer supporting Radio Frequency Identifier (ıRFID) tags, this can be detected and the information of the product can be viewed and printed. As

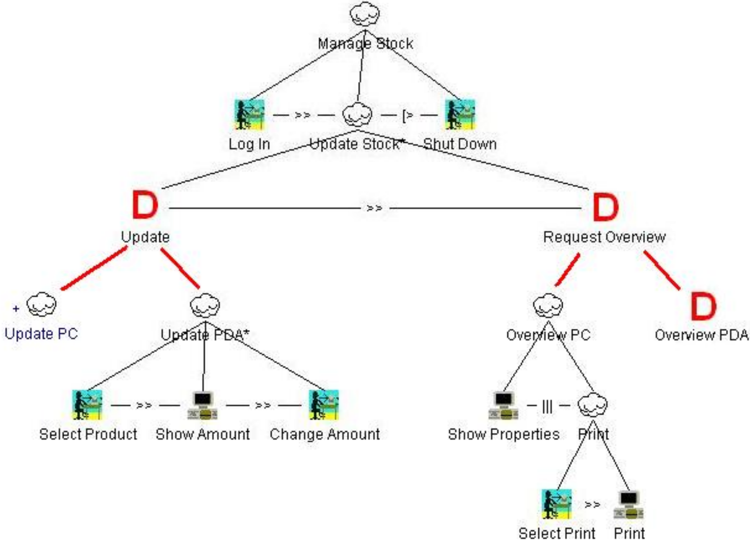


Figure 10.2: Context-Sensitive Task Model of the *Manage Stock* example

a result, the example contains two types of context denoted by the decision tasks: platform (*Update* and *Request Overview*) and location (*Overview PDA*). To link the context handler to the appropriate decision node, decision rules need to be attached to these nodes. Listing 10.3 shows an example for the *Overview PDA* task. In this case there will be a call for the *canPrint* function in the RFID Reader.

Listing 10.3: Decision rules for the *Overview PDA* task

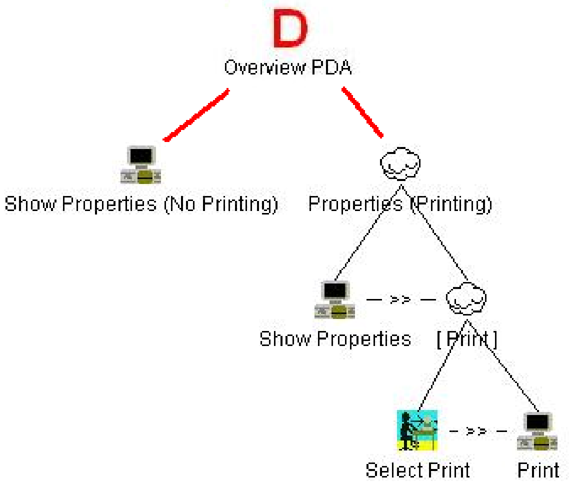


Figure 10.3: *Overview PDA* subtree

```
<decision>
   <cond type="equals">
      <value type="context">RFID:Reader:canPrint</value>
      <value type="boolean">true</value>
   </cond>
   <true platform="context">
         Show Properties (No Printing)
    </true>
    <false platform="context">
         Properties (Printing)
    </false>
</decision>
```

Listing 10.4: Decision XML for the *Use Properties* task

```
<decision>
    <cond type="equals">
         <value type="context">RFID:Reader:canPrint</value>
         <value type="boolean">true</value>
    </cond>
    <true platform="context">Show Properties (No Printing)</true>
    <false platform="context">Properties (Printing)</false>
```

```
</decision>
```

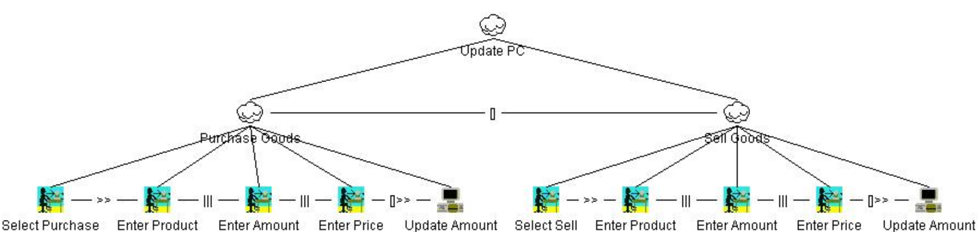


Figure 10.4: *Update PC* subtree

The first step to automatically generate the user interface is to convert the context-sensitive task model into a context-specific task model. This is why the condition in the decision XML has to be evaluated for each decision node and the decision node is replaced by its subtree which matches the current context. In the *Overview PDA* task example from figure 10.3, there will be an evaluation of the *canPrint* function. If the return value equals *true* the *Properties (Printing)* subtree will replace the decision node, else the *Show Properties (No Printing)* will. Figure 10.5 shows the context-specific task model in case of using the PC to change the stock amounts and the PDA to notify the employee within the reach of an RFID supporting printer.

The next step uses a custom algorithm (described in section 5.5.3) to calculate the enabled task sets:

$$
\begin{aligned}
ETS_1 = \;& \{Log\ In\} \Rightarrow P_{all} \\
ETS_2 = \;& \{Select\ Purchase(P_{pc}), Select\ Sell(P_{pc}), Shut\ Down\} \Rightarrow P_{pc} \\
ETS_3 = \;& \{Enter\ Product(P_{pc}), Enter\ Amount(P_{pc}), Enter\ Price(P_{pc}), \\
& Shut\ Down\} \Rightarrow P_{pc} \\
ETS_4 = \;& \{Enter\ Product(P_{pc}), Enter\ Amount(P_{pc}), Enter\ Price(P_{pc}), \\
& Shut\ Down\} \Rightarrow P_{pc} \\
ETS_5 = \;& \{Update\ Amount(P_{pc}), Shut\ Down\} \Rightarrow P_{pc} \\
ETS_6 = \;& \{Update\ Amount(P_{pc}), Shut\ Down\} \Rightarrow P_{pc} \\
ETS_7 = \;& \{Show\ Properties(P_{pda}), Shut\ Down\} \Rightarrow P_{pda} \\
ETS_8 = \;& \{Select\ Print(P_{pda}), Shut\ Down\} \Rightarrow P_{pda} \\
ETS_9 = \;& \{Print(P_{pda}), Shut\ Down\} \Rightarrow P_{pda}
\end{aligned}
\tag{10.1}
$$

P_x indicates on which platform the tasks can be executed. $x = all$ means the platform does not matter, and the task can be executed both on a PC or on

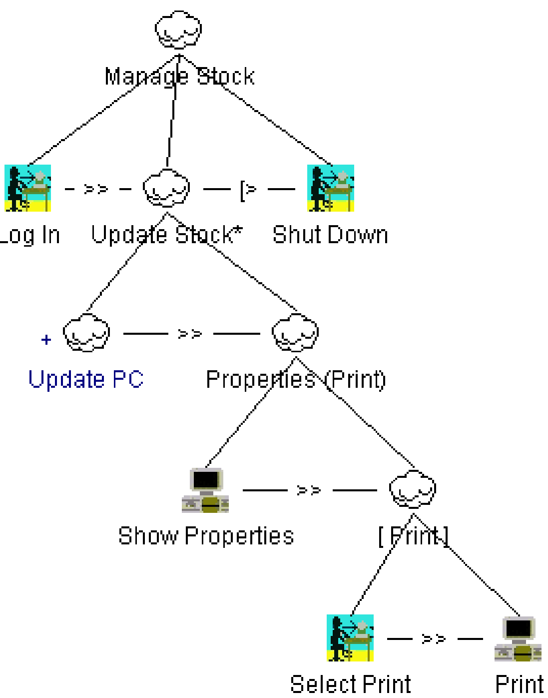


Figure 10.5: Context-Specific Task Model

a PDA. This example only contains tasks restricted to either a PC or a PDA because no enabled task set contains tasks marked P_{pc} and P_{pda}. Remark that the only difference between ETS_3 and ETS_4, and ETS_5 and ETS_6 is they are children from another task. Afterward, the dialog model (figure 10.6) is automatically extracted. Finally the actual user interface is rendered by the run-time environment. Figure 10.7 shows the dialog model with the rendered user interfaces.

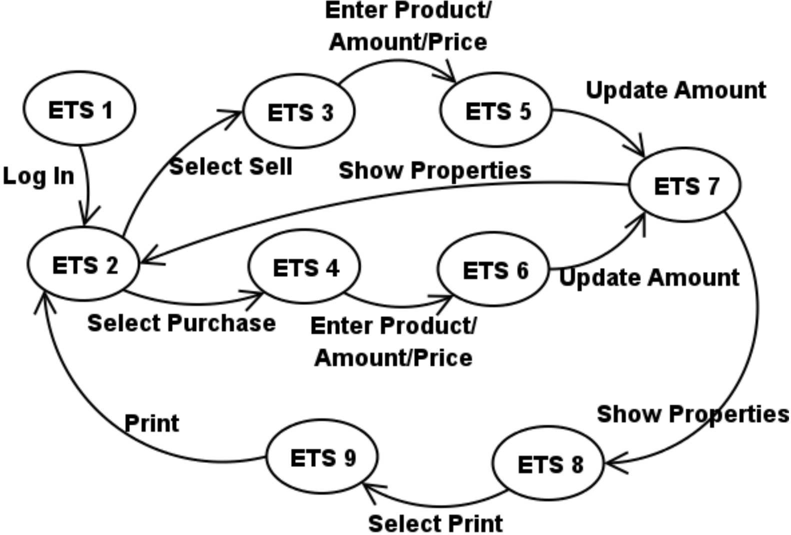


Figure 10.6: Dialog Model (The accept state caused by the *Shut Down* task is omitted to avoid cluttering the picture.)

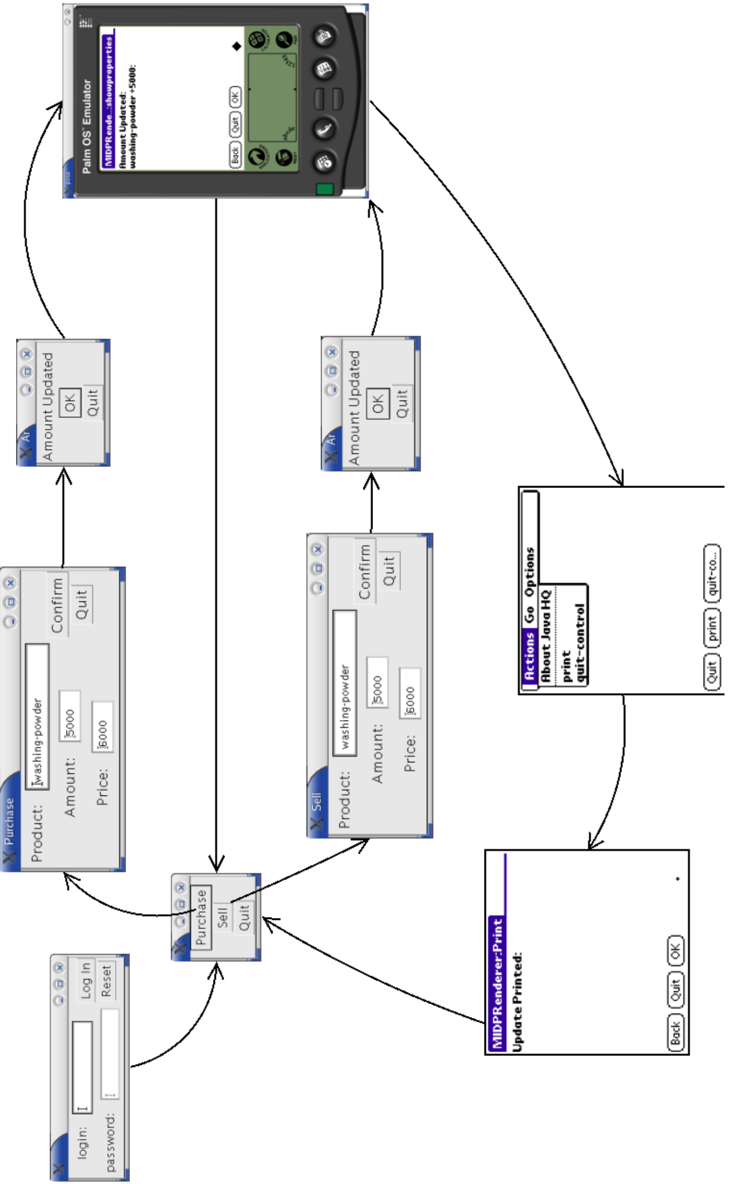


Figure 10.7: Dialog Model with the concrete dialogs

10.6 Discussion

This chapter shows how context information can be integrated in interface design to generate multi- and multiple-device user interfaces at run-time. The ConcurTaskTrees formalism is combined with decision nodes and rules to allow the user interface to adapt to the context while still being consistent w.r.t. the design. An important case is where the context can indicate the change in interaction device while executing a task. Our model allows this change by providing an appropriate dialog model including the transitions between dialogs on the same device *and* transitions between dialogs on different devices. The presentation model also supports dialogs that are distributed over several devices. The precondition to make this work is the context must be frozen from the start until the end of the main task.

The work presented in this chapter is preliminary work, built upon the results of the previous chapters. We are extending support for context-sensitive user interfaces with Model-Based User Interface Development by introducing a degree of ıdialog plasticity in [CLC04b] and making context explicit in the different models that are used in the Dygimes process in [CLC04a].

Chapter 11

Future Work

Contents

11.1 Dynamic Model-Based User Interface Development 186
11.2 Distributed User Interfaces 187
11.3 Next-Generation Widget Toolkits 187
11.4 Software Engineering 188

This chapter provides new research topics and questions that can be tackled in the near future. Referring back to the scenario's in chapter 1 we see there are still gaps that need to be filled up. The CoDAMoS project[1] is a Flemish project that targets exactly the kind of technologies that is necessary to realize the aforementioned scenarios. Within the EDM there is also ongoing research work that goes beyond the topics presented in this dissertation. This chapter gives an overview of the topics I consider to be the most important ones.

In short term a few improvements can be made to the Dygimes framework. The tools that are provided now are proof of concept tools and are not suitable for use in a real environment. The tools lack usability and do not provide enough feedback to the designer. Concerning the separate modules, the rendering engine needs to be expanded so it can express more divers user interfaces (see also chapter 9 for a possible expansion) and the layout management module needs to take into account user defined rules to make the final user interface more usable and visually pleasing. The framework should also

[1] CoDAMoS (Context-Driven Adaptation of Mobile Services) project IWT 030320, `http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/CoDAMoS/`

be expanded to include other models more easily. In spite of these improvements that would make our achievements more usable, Dygimes has proved to be suitable as a basis to implement new ideas that improve Model-Based User Interface Development. Chapter 10 already showed it could be easily extended to include "context" in the task model: [CLC04a, CLC04d, VC04, CLC04b] are all publications that rely on the Dygimes framework as the basis for their developments.

11.1 Dynamic Model-Based User Interface Development

Interactive software has escaped from the desktop computer and can accompany the user(s) everywhere they go. The user will no longer be the only entity influencing an interactive system, there will be others (unexpected) entities that influence the state and behavior of a system, human and non-human. There is a big challenge in finding appropriate ways for modeling these kind of systems, the application logic as well the interactive part. Context-sensitive systems will increasingly become important as software becomes more mobile and our environment gains in computing power and network communication possibilities. The next-generation Model-Based User Interface Development Environment should integrate context explicitly in the user interface design process, with a clean comprehensible definition of context.

Within the CoDAMoS project, an ontology for context was created that can serve as a basis for this integration [PVW+04]. The main difficulties start once context for a Model-Based User Interface Development Environment is defined: models should become fully dynamic. E.g. the task specification could change according to newly found context information, but how should these changes be anticipated at the design stage of an application. E.g. the execution of a task could become impossible because of changes in the environment of the user. Other models such as the dialog model, presentation model and even the domain model will also be influenced. We published some preliminary work in [CLC04b] and have another paper submitted [CLC04a]. This kind of work would support the design of pervasive interactive systems and enable us to create ambient intelligent environments from a user's point of view.

11.2 Distributed User Interfaces

In this dissertation we emphasized multi-device user interfaces, and even interfaces that could migrate from one device to another and adapt according to the target device. A user interface should be designed such that it could be easily used while being distributed over multiple (possibly communicating) devices. A user interface should take advantage of all the appropriate devices that are within the user's reach, and essentially be able to split itself into several parts so that each part could be allocated to the most appropriate device. There are many issues that have to be tackled here: define an optimal topology for a user interface distribution, create or use an environment that helps to harvest the different devices and supports transparent communication between devices, determine usability metrics for certain distribution topologies,...

Again, this kind of work would support the creation of interactive pervasive systems and ambient intelligent environments for the user's point of view. Some preliminary work from our research group concerning distributed user interfaces can be found in [VC04].

11.3 Next-Generation Widget Toolkits

The two previous goals are not possible without the support of a toolkit to actually realize the final concrete user interfaces. Before we can take the leap towards real distribution using multiple modalities and dynamic models, we should draw up what will be a next-generation widget toolkit (widget is used here in the broad sense, meaning a concrete interactor). At current, a great diversity can be detected that is too scattered to allow a designer to take advantage from all: physical or tangible widgets, speech widgets, GUI widgets,...

First thing is first: before physical widgets and the like come into play there will be some time to look into GUI and voice interfaces and how these can be adapted to the new needs. Traditional GUIs need to be enhanced with a new type of layout system that is device independent but also allows a smoother transition of (parts of) the GUI into a speech interface. The first step towards distribution is to make traditional GUIs "distribution-aware" so they can stretch over several screens for example. Usability borders of an adaptive interface is also an aspect that will gain importance: as the requirement for plasticity of the user interface presentation rises, we need to decide on the borders of this plasticity w.r.t. the usability of the user interface. We did some very preliminary work on the last subject in [LC04a].

11.4 Software Engineering

Since the OMG introduced Model Driven Architecture (MDA) as the new process for creating multi-platform software, the urge to integrate Model-Based User Interface Development and traditional software engineering has only become more prominent. The IFIP working group 2.7 / 13.4 on User Interface Engineering[2] has organized several workshops around this topic. Model-Based User Interface Development and MDA seems to fit nicely together: e.g. UsiXML [LVM+04b] is an example that uses a transformation approach just like MDA does. But there are many issues that need to be resolved: the fact there is no environment that is widely accepted and integrates Model-Based User Interface Development and a general software engineering tool is an indication there is still some work that needs to be done.

Suggestions here are to define a standard model for Model-Based User Interface Development and relate it to software engineering processes such as MDA or the Rational Unified Process (RUP). This means the different stages of a process (requirements, analysis, design, testing, deployment), structures (waterfall model, star model, transformational development,... [Som04]) and notations (UML, Booch, Rumbaugh, ...) need to be aligned with the different models and model notations from Model-Based User Interface Development. To allow a wide adoption an integrated case tool combining these techniques is a must.

[2] http://www.se-hci.org

Chapter 12

Conclusions

This dissertation is concluded with a short summary of the work and an overview of our achievements.

12.1 Model-Based User Interface Development

Model-Based User Interface Development has been an active research subject since many years, and has become more relevant each year. Since the rise of a new class of computing devices this approach has gained even more importance. It offers us a method to cope with a new set of requirements that are subject to continuous changes. Since the explosion of different kinds of devices this has become one of the main problems in user interface design: minimizing the effort while supporting a maximal wide range of different devices. In this dissertation we have presented a new system for Model-Based User Interface Development: Dygimes. This systems supports and can execute selected models (abstract and concrete) to create an interactive system for multiple devices.

Dygimes provides a simple and clear process to create multi-device and multi-context user interfaces. The design of the user interface starts with the task model as the central model. Once the task specification is created in the appropriate notation other models are related with this model or are generated on the basis of this model. By using several algorithms the consistency between the different models is ensured, and the user interface designer will end up with an interactive system that aligns perfectly with the predefined task

specification. This does not mean the designer has no further influence: the specific user interfaces, referred to as user interface building blocks, are still being created by the user interface designer. With SEESCOA XML, layout can be specified by using constraints that cover the relations between interactors insided buiding blocks as well as the positioning of different building blocks with respect to each other. Depending on the User Interface Description Language being used the designer can also specify rendering hints that influence the appearance of the user interface. Finally, the widget mappings can change the final user interface by changing the selection of concrete interaction objects for each abstract interaction object.

The techniques introduced in this text are becoming more common to create user interfaces because of the changing requirements. Even though not always as visible, task-based design gains popularity and multi-device user interface creation is a must. The next step is to support context-sensitive user interfaces.

12.2 Achievements and Main Contributions

This section tries to summarize our achievements and main contributions to this field of research. In section 1.2 we wrote down our motivation and aims for conducting this work. We recapture the five challenges from section 1.2 and evaluate our achievements and contributions for each of these challenges:

Challenge 1 Task-Centered Interfaces: a great deal of effort has been put into this challenge. Our approach allows a task centered design of the user interface, and ensures consistency of all final user interfaces with the task specification.

Challenge 2 Multi-Platform Support: it was our primary goal to ensure a user interface designer only had to design the user interface once instead of doing a redesign for each platform that should be supported. We think we have succesfully done this by adding new techniques along the way starting from the task model to the concrete user interface: we created an XML-based User Interface Description Language, provided device independent layout management and introduced support for an arbitrary application domain,... Because of the combination of using an XML-based User Interface Description Language with a flexible layout management algoritme a user interface becomes more "plastic" so it can adapt better to the target platform. Without restrictions to a specific type of application domain it becomes easier to deploy the user interface

and allow it to access the application logic remotely or locally depending on the situation.

Challenge 3 Interface Tailoring: Only minor effort went into this subject. Depending on the XML-based User Interface Description Language that has been used the designer can specify "rendering hints" that influence the appearance of the final user interface. E.g. Seescoa XML has a separate subsystem that allows interface tailoring to a certain degree and UIML can make use of style properties to taylor the user interface.

Challenge 4 Multi-Modal Interfaces: besides some tests we did to support speech-driven dialogs in SEESCOA XML, no efforts were done to create a full multi-modal output model for Dygimes. We are convinced other modalities require different designs methodologies, and only parts of the designs for the different modalities can be based on the same design specifications. Other initiatives (e.g. Teresa) proof the task model can be a good basis to develop a multi-modal user interface, although the design for different modalities has to be split up at a certain more concrete stage. It is unclear how different modalities can be mixed together however.

Challenge 5 Support for Context-Sensitive Interfaces: the Dygimes framework allowed us to build extensions that support the design of context-sensitive user interfaces. We already created a prototype design environment that integrates context and design *at realtime*. Context can be gathered by sensor data or simulated, and the designer can immediately see how this data influences the different models and the final user interface *while* she/he is designing the user interface.

12.3 Scientific Contributions and Publications

The work presented here is built upon four years of research. This section provides an overview of the publications that reported on this research, and were presented in international scientific conferences.

[LC01], 2001 *Kris Luyten and Karin Coninx. An XML-based runtime user interface description language for mobile computing devices. In Johnson [Joh01], pages 17–29*

This paper laid out the foundations of our current work and introduced the multi-device user interface creation together with XML-based High-Level User

Interface Description Language in our research. As such its content contributed to chapter 6.

[LLCR03], 2002 *Kris Luyten, Tom Van Laerhoven, Karin Coninx, and Frank Van Reeth. Runtime Transformations for Modal Independent User Interface Migration.* Interacting with Computers*, 15(3):329–347, 2003*

This journal article illustrates next step we took: investigate the feasibility to support multiple modalities besides multiple devices.

[LVC02], 2002 *Kris Luyten, Chris Vandervelpen, and Karin Coninx. Migratable User Interface Descriptions in Component-Based Development. In Forbrig et al. [FLUV02], pages 62–76*

This paper went beyond the multi-device concepts and introduced an application model as a set of components and distribution of the user interface and application logic over different devices. It serves as a basis of chapter 8.

[LCCV03], 2003 *Kris Luyten, Tim Clerckx, Karin Coninx, and Jean Vanderdonckt. Derivation of a Dialog Model for a Task Model by Activity Chain Extraction. In Jorge et al. [JNF03], pages 203–217*

After introducing a domain model and distributed interfaces in the previous model, the next challenge was to take into account the necessary temporal rules to generate a dialog-based user interface. We designed and developed an algorithm that extracts a dialog specification from a task specification based on the temporal relations between the tasks. It is the basis of chapter 5.

[CLV+03], 2003 *Karin Coninx, Kris Luyten, Chris Vandervelpen, Jan Van den Bergh, and Bert Creemers. Dygimes: Dynamically Generating Interfaces for Mobile Computing Devices and Embedded Systems. In Luca Chittaro, editor,* Mobile HCI*, volume 2795 of* Lecture Notes in Computer Science*, pages 256–270. Springer, 2003*

A complete overview of our process is published in this paper, in particular how it supports mobile systems. It is the basis of chapter 4.

[LC04b], 2004 *Kris Luyten and Karin Coninx. Uiml.net: an Open Uiml Renderer for the .Net Framework. In Limbourg et al. [LJV04]*

Since the target platforms for our user interfaces expanded beyond embedded systems and mobile computing devices, the need for a new kind of XML-based User Interface Description Language arised. It needed to be a highly extensible language. We choose to implement a UIML Renderer and presented our results in a paper. This paper is the basis of chapter 9.

[CLC04c], 2004 *Tim Clerckx, Kris Luyten, and Karin Coninx. Generating Context-Sensitive Multiple Device Interfaces from Design. In Limbourg et al. [LJV04]*

This paper adds context-awareness to the approach which is described in the previous paper. It does this on a task level, where context decision nodes are used to make it context-aware. It is the basis of chapter 10.

Besides these international scientific publications we published a technical report on device independent layout techniques in early 2003. This publication is the basis of most of chapter 7:

[LCC03], 2003 *Kris Luyten, Bert Creemers, and Karin Coninx. Multi-device Layout Management for Mobile Computing Devices. Technical Report TR-LUC-EDM-0301, Expertise Centre for Digital Media – Limburgs Universitair Centrum, Belgium, 2003*

In addition, based on the experiences obtained by the research contributions presented in this chapter, we organized a workshop at AVI'2004:

[LALV04], 2004 *Kris Luyten, Marc Abrams, Quentin Limbourg, and Jean Vanderdonckt, editors.* Developing User Interfaces with XML: Advances on User Interface Description Languages. *Sattelite workshop of Advanced Visual Interfaces (AVI) 2004, Expertise Centre for Digital Media, 2004*

Some XML-based User Interface Description Languages that were presented on this workshop are also discussed in chapter 3.

12.4 Concluding Remarks...

In 2001 the research group "Gebruikersgerichte Systeemontwikkeling" (User-Centered System Development) was created and lead by prof. dr. Karin Coninx, based on the topics that are presented in this dissertation. Now, in 2004, there are three more PhD students doing research in this group and we are involved in several regional and European projects. The new PhD studies in our group have come into existence part because of the dissertation you just have read: unlike what we thought in 2001, this research topic offers many new challenges that can not possibly be done in one dissertation. Chapter 11 offered an insight in what is going on now in our research group. Model-Based User Interface Development is a reoccurring theme in all of them: in our work we combine the old ways with the new ones and try to generate solutions for existing problems with this combination.

Appendix A

Scenarios

A.1 Scenario 1: Teaching with Technology

A university decides to install projectors on the ceiling of every classroom. They will be using the system described in this dissertation to make the projectors accessible to different members of the teaching staff and maintenance personnel. All of the employees of the university have a PDA device (e.g. a Palm device), which can receive data over a infrared connection. Alternatively, a radio-based Bluetooth link between the PDA and the projector can be used. The PDA will be used as a remote control for the projector. During the first class of Thursday, professor Wasaname will have to use the projector for her lectures in HCI. She walks into the classroom and transmits her slides, stored on the PDA, to the projector using the infrared connection. Let us assume the projector knows the slide format and can store and project these. After she has finished transmitting the slides she indicates on her PDA she wants to control the projector. Because her user profile is stored on the PDA, it infers she only wants to cruise through the slides, maybe zoom in on some details and make some annotations, but nothing more. She is not interested in configuring the projector settings like changing the resolution or the color settings. Using this knowledge the PDA "asks" the projector to transmit only those parts of its user interface professor Wasaname is interested in. The projector serializes that part of the user interface and passes it to the PDA device, using the infrared connection. The professor can now project the slides using the PDA as a remote control.

During her first class, professor Wasaname notices the bad resolution and

brightness of the projector. After her lesson, a member of the maintenance personnel is asked to fix the problem. The diligent responsible man gets right to the classroom and indicates he wants to use the projector. Looking at his profile, the PDA notices this person is mainly interested in the configuration possibilities of the projector, and asks the projector to only transmit that part of the user interface dedicated to that task. Using his PDA the maintenance man adjusts the brightness and resolution of the projector to a satisfying level.

A.2 Scenario 2: Mobile Communication

Instant Messaging (IM) is one of the most popular forms of communication on the Internet. When the user is mobile, sometimes IM over the Internet is not possible and the user will use its mobile phone. This scenario describes extended IM, where the location of the user and the device he/she uses is transparent for the IM service. IM will become context-sensitive, observing information in its environment and adapting according to this information.

The scenario in this section is developed in cooperation with the Research and Development group of Alcatel[1] as part of a CoDAMoS deliverable.

This evening a soccer match is scheduled between RC Genk and RSC Anderlecht. Dave, Evy and Rob are three soccer fans and are interested in going to the match. They are also three friends, although Dave and Evy are supporters of RC Genk and Rob is a supporter of Anderlecht. Since the match is being played in Anderlecht, they are arranging how to get there using the Instant Messaging software on their desktop PC. The match starts at 8 o'clock, so in a group chat they agree to leave at 6 o'clock and Dave will drive.

Dave planned to drop those music sticks he lent from her off at Evy's and asks whether she knows when she will arrive home so he can come by. Evy commands by the use of her voice her car to calculate the approximate time she will arrive at her home. The car's software uses its internal GPS and route-planner and calculates the result. It notifies Evy with the result and Evy commands the car to send this result to Dave and to notify him when she really arrives at home (in case the estimation was not correct by some chance). The car can use the software of the mobile phone to perform this functionality.

[1] Alcatel in Belgium is part of the worldwide Alcatel group. Alcatel designs, develops and builds communication networks that allow telecommunications operators and companies to transmit all types of content (voice, data or multimedia) to their customers worldwide. `http://www.alcatel.be/`

Evy arrives at her home, and meets with Dave there to receive her music sticks. Since Evy has hurt her foot and has to stay at home this evening, she will watch the soccer match on her television, while staying in contact with Rob and Dave. At home, she notices a virtual indicator, coming from Rob, on the television inviting her to watch the match together. He will be at the stadion and stay in touch with her through Instant Messaging, as is Dave.

Rob and Dave both carry a mobile phone with a built-in camera and IM software on it. In their collars they have a microphone allowing them to speak to each other without taking the phone out of their pockets. When Evy sits in front of the television and turns it on, she won't be disconnected with her friends if she does not want to. Her mobile phone and set-top box synchronize their contact lists, taking into account the privacy settings of each contact. When she puts on the channel that broadcasts the soccer match, and gets a short update on-screen of her friends watching the same channel. They are all presented by their personal avatars, blended with the television screen.

She wants to make a prediction about the game and put it on a virtual whiteboard so her friends can see it. She can see the predictions her friends made too. After the match the people who predicted the correct result get a free drink at the pub where she always meets with her friends.

But Evy watches the soccer match together with her mother, and she does not want to distract her with the messages she exchanges with Dave and Rob. So she gets her PDA and the set-top box will open a channel to the PDA so the messages sent to her will only be shown on the PDA and not on the television screen. When her mother enters the room, her "private" contacts disappear from the screen and only the contacts that are shared with her mothers contact list are still visible. The avatars disappear of the screen so her mother does not get to see them. The PDA takes over functionality previously offered by the set-top box. This way she can share the television screen with others while her private communications will be automatically redirected to her PDA. The set-top box serves as a kind of local messaging service for the family "participants" of the current television show.

Once Rob and Dave enter the sports ground (each on the other side, because they are fans of different soccer teams) their mobile phones notice they are inside. The contact list of Evy will be automatically updated with entries for Dave and Rob, because they are now also participating (as viewers) at the same event. From now on the group can start sharing experiences with each other. If she wishes Evy gets to hear the crowd in real-time through the voice communication from where Dave and Rob are standing. The snapshots they take with their mobile phones' built-in camera can be browsed on the PDA

by Evy.

Evy goes to the kitchen for a drink, but she is a really big fan so she does not want to miss one bit of the match. While she moves away, the set-top box detects she is not watching anymore. However, her PDA still is in watching mode. So available information like the current score and who is in ball-possession will be transmitted to the PDA in real-time. The PDA is not programmed beforehand to visualize this information. It will receive a User Interface Description that allows the PDA to render the content in an appropriate form. The User Interface will be merged with the communication channel for IM, so both IM and observing the state of the match will be possible at the same time. Even if the connection between the PDA and the set-top box fails, she does not have to worry: every bit of the broadcast is recorded digitally so she can replay the stuff she missed.

Anderlecht had just scored a goal and Evy is sure it was off-side. Only, the television images are not clear enough, so she asks Rob whether he knows for sure. Luckily, Rob has recorded a small image and he sends it over to Evy. While the set-top box processes the video, it checks the timestamps in the video format. Evy replays the goal on the big television screen while she lets Rob's video play on her PDA. Both the set-top box and the PDA synchronize so she gets to see both views (different angles) at the same time. She is a little bit disappointed: it was certainly not off-side.

A.3 Scenario 3: A Mobile Tourist Guide in a Museum

A group of teenagers will visit the Gallo-Roman Museum of Tongeren as part of their history course together with the teacher for this course. They will get a guided tour through the museum by an experienced guide. This is a common activity for the class, and they are required to assemble information from these activities. The information they assemble is part of their exam and it will help them in getting high scores. The whole museum visit is about *discovering history*; getting to know the historic facts by "playing" archaeologist.

Luckily education has changed a bit. Every student has its *personal information space* that includes all kinds of personal devices they carry with them and the information contained on this set of devices. All students have a Personal Digital Assistant (PDA) that allows them to capture data, carry digital information, process and visualize information and communicate with each other. A PDA is every kind of device that can be hold in one hand, has

input and output capabilities, and can load new software while it is running. In this context, a mobile phone can be used as a PDA. For their museum visit they are required to carry their PDA with them.

When the class arrives at the museum, they will be split in several groups. The students in these groups can cooperate to find (*discover*) useful information and process it accordingly. To support easy information exchange the PDAs of a group are logically connected; they can access each others information. Once the teenagers enter the museum entry hall, they are registered by the museum administration. Since the PDA is a personal device it has the user profile of the owner stored. Their PDAs send the parts that are not private of their profiles to the museum over their wireless connection for registration purposes (this kind of data could also be used for data-mining), in return their PDA is connected to the wireless LAN of the museum. The museum and visit *specifications* are downloaded onto the PDA automatically. A general micro-interpreter transforms these specifications in a user interface on the PDA. The specifications contain a task specification (what can be done in the museum, what are the goals of the visit), a domain specification (an ontology that specifies the essential objects in the museum and their relations), a presentation specification (the way the information is shown to the user) and an interaction specification (what kind of data exchange exists between the different specifications and between the PDA and the museum system). Other specifications that contribute to the complete interface are also transmitted, but are of minor importance.

The teacher checks whether everyone is registered in the museum and ready to start, after which the tour begins. The visit to the museum is a combination of a kind of discovery game and a guided tour. The goal of the visit is to discover the meaning of the Dodecahedra artifact. Throughout the tour there will appear questions on the screens of their PDA, which leads the group to a piece of information. To answer the questions correctly they will need to pay attention to what the guide says, and to what the artifacts they can view tell them. If they need more information of an artifact the visitor only has to move closer to the artifact: the PDA sensor will sense the proximity of the artifact and show more information about that particular artifact. The communication works in two ways: the user can send questions to the artifact by hand (by voice and speech-recognition) and whenever the information that could answer this question is stored with the artifact it will be sent to the PDA. This way the user "gathers" useful data about the different artifacts, which can help her/him to solve the mystery of the Dodecahedra. If one member of the group solves a question, the next piece of information that

is necessary is unlocked for the whole group. A distributed system takes care of the communication between the different PDAs.

When the users walk through the museum, both the user interface and the data in the user interface will adapt. The museum is divided into different areas, where each area represents another period in history. When the users enter another area, the system sends another presentation specification to the PDA, and the user interface will be adapted at runtime to reflect the historical changes (e.g. from the human being a collector to a hunter).

Guiding a visit of a whole class of teenagers is not an easy task: the guide needs to control the group and the PDAs that are used in the group. The guide needs to avoid the visitors can try to unravel the next part of the Dodecahedra mystery before the group has moved on to the next part of the exhibition. The guide can control the visitors PDA to ensure the visit stays focused on the artifacts that are nearby the class. With an appropriate selection menu, the guide can disable and enable parts of the user interface and the data that is shown to the users.

When a guide approaches information panels (output screens like television screens, whiteboards, flat screens, projector screens,. . .) a part of the interface of the guide will change into a remote control for these information panels. Only the PDA of the guide will download the user interface for the information panel, because the user interface is only available for PDAs with the appropriate user profile. The guide can select a set of information she/he wants to show on the information panel.

During their visit all PDAs capture the route that has been taken, the information that has been communicated and the answers on the questions the students have solved on the device. Cameras in different corners of the museum capture the moments a guide is explaining something and these captured movies are also transferred to the PDAs. This way each student has a completer overview and reference from their visit.

At the end of the tour each group has an overview of their answers on the questions and the list of their observations they made during the tour, both visualized on their PDA. Now it's time for each group to suggest an answer for the Dodecahedra mystery. Their suggestions are compared to some of the existing theories that are revealed at the end of the tour by the guide (by projecting possible explanations on the information panel at the exit). By looking through the sequence of questions and answers initiated by the students, they can evaluate how good of an archaeologist they would be.

A.4 Technological Challenges

The first scenario (A.1) can be situated as our goal when we started to develop this approach, we wanted to be able to support:

- Migratable User Interfaces: interfaces that can be transferred from one device to another one.
- Adapting the user interface to the host device.
- Transferring control over a device to another device, supporting the users task.

The second scenario (A.2) introduces ambient intelligence; "context" becomes more important here:

- The user interface becomes context-aware.
- Communication between devices and devices, people and devices, and people and people becomes more important.
- Intelligent integration of devices in the environment
- Multi-modal interfaces

Finally, the third scenario (A.3) shows the importance of social networking that should be taken into account when designing user interfaces. These three scenarios illustrate the goals that were set in several of our research projects. This dissertation selects some of the technical requirements that are necessary to develop scenario one and two. The third scenario mainly adds some social aspects to the ones that are already mentioned in the previous two scenarios.

Appendix B

Nederlandstalige Samenvatting (Dutch Summary)

B.1 Inleiding

Door de toenemende diversiteit van beschikbare "programmeerbare" apparaten wordt het ontwikkelen voor interactieve systemen voor deze toenemende diversiteit complexer. Gebruikersinterfaces dienen herbruikbaar te zijn voor verschillende apparaten die telkens andere, specifieke eigenschappen bezitten. Er is een nood aan een methodologie die het ontwerp, de ontwikkeling en de verspreiding van dit soort gebruikersinterfaces. We zullen in de rest van dit hoofdstuk het over "multi-apparaat interfaces" als we het hebben over gebruikersinterfaces die zonder enige manuele aanpassing kunnen opnieuw gebruikt worden op diverse, onderling verschillende apparaten.

In deze dissertatie wenden we model-gebaseerde interface ontwikkeling aan ter ondersteuning van multi-apparaat interfaces. Model-gebaseerde interface ontwikkeling was reeds een gekende manier van "traditionele" interface ontwikkeling in het begin van de jaren 90, en blijkt tevens uitermate geschikt te zijn voor het ontwerp van multi-apparaat interfaces [EVP01, SLV02, PS02, CLV$^+$03, CLC04b, MPS04].

Szekely identificeerde vier uitdagingen in [Sze96] voor model-gebaseerde interface ontwikkeling:

- Uitdaging 1: Taak-gebaseerde interfaces.
- Uitdaging 2: Ondersteuning voor een diversiteit aan apparaten.
- Uitdaging 3: Verfraaien van interfaces.

- Uitdaging 4: Multi-modale interfaces.

De eerste drie uitdagingen worden in deze dissertatie behandelt, waarbij het zwaartepunt ligt op uitdagingen één en twee. We voegen een vijfde uitdaging toe aan deze vier uitdagingen, waaraan we ook de nodig aandacht zullen besteden:

- Uitdaging 5: Ondersteuning voor context-gevoelige interfaces.

Sectie B.9 zal wat dieper ingaan op deze vijfde uitdaging.

Het doel van deze thesis bestaat eruit om een methodologie en raamwerk te ontwikkelen die de creatie van multi-apparaat interfaces ondersteunt in alle stadia van de ontwikkeling (design, implementatie, deployment) en een omgeving aanbiedt om deze interfaces uit te voeren. Hierbij wordt er aandacht besteed aan de herbruikbaarheid van de verschillende technieken die voorgesteld worden.

B.2 Model-gebaseerde Gebruikersinterface Ontwikkeling

Vooraleer dieper in te gaan op de methodologie die we zullen voorstellen, bekijken we wat model-gebaseerde interface ontwikkeling exact voorstelt. Vanwege de grote diversiteit is er geen algemeen aanvaarde afbakening van wat model-gebaseerde interface ontwikkeling juist betekent en hoe "model" kan gedefinieerd worden. Er is echter wel een consensus over de concepten die algemeen gebruikt worden: een model kan bekeken worden als een verzameling onderling gerelateerde informatie die een bepaald aspect van een interactief systeem op een abstracte manier beschrijft, waarbij de laag-niveau details achterwege gelaten worden maar de belangrijke "karakteriserende" details opgenomen worden in het model.

Vele van de eerste model-gebaseerde aanpakken borduurden verder op een meer formele basis [LS96b, SLN92, SSC+95], maar dit bleek onvoldoende bruikbaar te zijn voor de modale interface ontwerper en/of ontwikkelaar. De declaratieve aard van model-gebaseerde interface ontwikkeling [Pin00] resulteert echter reeds in een "meer formele" basis van de gebruikte notaties. We zien dan ook dat een informele notatie, gecombineerd met een formele basis de voorkeur heeft [MPS04, MPS02, LVM+04b, LV04, LVM+04a, CLV+03, CLC04b].

Er zijn verschillende algemeen aanvaarde modellen, zoals het taakmodel, het dialoogmodel, het presentatiemodel, het domeinmodel, het gebruikersmo-

del en het applicatiemodel. Elk van die modellen beschrijft een belangrijk aspect van het interactieve systeem: zo beschrijft het taakmodel het doel dat de gebruiker kan bereiken met het systeem en welke taken er door het systeem ondersteund worden. Het dialoogmodel is sterk gerelateerd en beschrijft de manier waarop er met de interface gewerkt kan worden: zo kan de navigatie doorheen verschillende onderdelen van een interface beschreven worden met een dialoogmodel. Het presentatiemodel specificeert wat er juist zal getoond worden aan de gebruiker; het definieert de structuur en inhoud van de dialogen die aan de gebruiker gepresenteerd worden. Deze drie modellen staan ook centraal in onze aanpak, *Dygimes*, die we in de volgende sectie uitgebreider zullen bespreken. De verschillende modellen kunnen onderling gerelateerd en/of getransformeerd worden [LVS00, CLC04d, VLF03] en vormen zo een krachtig geheel om een gebruikersinterface te specificeren waarbij aan de hand van de modellen kan geverifieerd worden dat deze aan de vooropgestelde eisen voldoet.

B.3 Dygimes: Dynamische Generatie van Interfaces voor Mobiele en Ingebedde Systemen

Dygimes staat voor Dynamische Generatie van Interfaces voor Mobiele en Ingebedde Systemen [CLV+03] en is een raamwerk en omgeving voor het bouwen en uitvoeren van multi-apparaat interfaces. Het steunt op de principes van model-gebaseerde interface ontwikkeling: er wordt namelijk gebruik gemaakt van het taak-, het presentatie- en het dialoogmodel om de interface te ontwerpen. Verder voorziet Dygimes een runtime omgeving die deze modellen kan uitvoeren: m.a.w. die de gebruikersinterface beschreven door deze modellen enkel aan de hand van deze modellen kan genereren en tonen. Bij Dygimes hoort een proces dat gevolgd kan worden om de interface te creëeren. Dit proces staat beschreven in figuur 4.1 op pagina 54; het gedeelte dat binnen de stippellijnen staat wordt automatisch afgehandeld, het gedeelte erbuiten vergt input van de interface ontwerper en/of ontwikkelaar.

Voor het taakmodel maken we gebruik van de ConcurTaskTrees (CTT) notatie [MPS02, Pat00]. Dit is een hiërarchisch taakmodel met een grafische notatie. CTT ondersteunt de temporele operatoren die ook te vinden zijn in LOTOS [LFHH91]. Het feit dat er concurrente taken toegestaan worden resulteert in het concept van Enabled Task Sets (ETS). Een ETS is een verzameling van taken die tijdens dezelfde tijdsperiode actief kunnen zijn. In onze aanpak wordt elke ETS op een dialoog met de gebruiker gemapt.

Het dialoogmodel wordt opgebouwd door een automatische transformatie die we uitvoeren op het taakmodel [LCCV03]. Met een zelf geschreven algoritme berekenen we de ETSs en leggen een volgorde in de tijd op aan de ETSs. We maken hiervoor gebruik van een staten-transitie netwerk: de ETSs zijn de staten in het netwerk en de transities tussen de staten worden gedetecteerd door een algoritme dat de temporele relaties uit het taakmodel verwerkt.

Het presentatiemodel in Dygimes wordt gespecificeerd aan de hand van een XML-gebaseerde hoog-niveau gebruikersinterface taal. Secties B.5 en B.8 geven twee mogelijke talen die in het Dygimes proces en raamwerk hiervoor gebruikt kunnen worden, namelijk SEESCOA XML en UIML. In het geval van SEESCOA XML kan er gebruik gemaakt worden van een flexibel layout management algoritme dat gebruik maakt van spatiële constraints (beperkingen). Zo wordt een gebruikersinterface voldoende rekbaar om gebruikt te worden in een multi-apparaat omgeving. Figuur 7.3 op pagina 124 toont hoe dezelfde gebruikersinterface-beschrijving kan gebruikt worden op verschillende apparaten: hiervoor dient er geen manuele tussenkomst meer te gebeuren.

Een specifiek, voorgedefinieerd applicatiemodel wordt niet gebruikt in Dygimes. Er wordt echter wel een "open protocol" voorzien vanuit Dygimes om met verschillende soorten applicatiemodellen te werken. Zo kan er gebruikt gemaakt worden van het direct oproepen van code, via XML-RPC code oproepen of van webdiensten [VLC03b]. Figuur 4.5 op pagina 62 toont hoe de link met het applicatiemodel op een transparante en locatie-onafhankelijke manier kan gemaakt worden. Met dit systeem kan de applicatielogica zowel lokaal als van op afstand door de interface gebruikt worden.

Voor de verschillende stadia in het Dygimes gebruikersinterface creatie proces, zijn er prototype applicaties gebouwd die telkens (een deel van) een stadium ondersteunen. Zo kan men gebruik maken van de volgende applicaties:

- Een applicatie die het taakmodel kan annoteren met gebruikersinterface bouwblokken (in de vorm van herbruikbare SEESCOA XML documenten, zie figuur 4.4 op pagina 59).
- Een applicatie, UiBuilder, die SEESCOA XML documenten kan renderen voor een bepaald platform (zie figuur 6.3 op pagina 109).
- Een applicatie waarmee de spatiële constraints voor een gebruikersinterface bouwblok kunnen gespecificeerd en getest worden (zie figuur 4.7 op pagina 67).
- Een applicatie, TaskLib, die een taakmodel omzet naar een dialoogmodel.

De volgende secties zullen meer informatie verstrekken over hoe de modellen respectievelijk gebruikt worden in ons systeem.

B.4 Modellen voor Interface Ontwerp voor meerdere Apparaten

Het taakmodel staat centraal in onze aanpak. Alvorens andere artefacten worden gecreëerd wordt er een taakspecificatie gemaakt in de CTT notatie. Deze taakspecificatie wordt vervolgens geannoteerd met gebruikersinterface bouwblokken. Dit betekent dat het taakmodel en (gedeeltelijk) het presentatiemodel door de ontwerper gemaakt worden. De geannoteerde taakspecificatie kan dan verder getransformeerd worden in een dialoogspecificatie, die vervolgens dan gebruikt kan worden om de interface effectief af te beelden.

De transformatie van een taakmodel naar een dialoogmodel is een complex proces en verloopt in verschillende stappen:

- De ETSs worden uit de taakspecificatie afgeleid, en vormen de dialogen van het dialoogmodel.
- De initiële ETS (waarmee het programma van start gaat) wordt gedetecteerd.
- De transities tussen de verschillende dialogen worden berekend uit de temporele relaties die voorkomen in het dialoogmodel. Vanaf dit moment wordt het dialoogmodel voorgesteld als een volledig staten-transitie netwerk.
- Vanuit het dialoogmodel kan met behulp van het presentatiemodel een gebruikersinterface gegenereerd worden.
- De designer kan veranderingen aanbrengen aan het dialoogmodel door heuristieken erop uit te voeren [CLC04d], de samenstelling van ETSs handmatig te veranderen,. . .

Figuur 5.1 op pagina 74 toont de verschillende stappen die genomen worden.

B.5 Presentatie van de Gebruikersinterface

De presentatie van de gebruikersinterface wordt in Dygimes verzorgd door een XML-gebaseerde hoog-niveau gebruikersinterface beschrijvingstaal. SEESCOA XML is een XML-gebaseerde interface beschrijvingstaal, ontworpen

op het einde van het jaar 2000 als een simpele, leesbare hoog-niveau specificatie taal om overdraagbare gebruikersinterfaces voor ingebedde systemen te bouwen [LC01, LLCR03, VLC04]. De taal werd ontwikkeld tijdens het IWT/STWW SEESCOA project, en werd gebruikt als eerste XML-gebaseerde beschrijvingstaal. Een volledig schema voor deze XML taal is afgebeeld in listing 6.1 op pagina 101. Voorbeelden van het gebruik van dit schema kan men vinden in listings 6.2 op pagina 103, 6.3 op pagina 105, 6.4 op pagina 106, 6.5 op pagina 111 en 6.6 op pagina 111. Merk op dat SEESCOA XML een hiërarchische structuur heeft bestaande uit groepen (de group tag), die zelf groepen of interactoren kunnen bevatten. Groepen zijn verzamelingen van elementen die logisch samen horen in de interface.

Om gebruik te kunnen maken van de SEESCOA XML taal werd er een Java-gebaseerde interface renderer ontwikkelt: de UiBuilder renderer. UiBuilder kan SEESCOA XML documenten omzetten naar Java AWT, Java Swing, Java kAWT, HTML en Java MIDP gebruikersinterfaces. Figuur 7.3 op pagina 124 toont hoe eenzelfde SEESCOA XML beschrijving resulteert in een bruikbare interface op meerdere apparaten. In een experiment werd de taal zelfs gebruikt om interfaces in een virtuele omgeving aan te spreken [LLCR03]: een interactie component uit de virtuele omgeving (in ons experiment een joystick) kon omgezet worden in een SEESCOA XML beschrijving zodanig dat een interactief element kon migreren van in een virtuele wereld naar een mobiel apparaat.

Een belangrijk onderdeel is de gebruikersinterface laten communiceren met de echte applicatielogica. Hiervoor voorzag SEESCOA XML een "open protocol" dat toelaat om de renderer en de XML-taal uit te breiden met een willekeurige manier om met applicatielogica te communiceren. Het klassediagram in figuur 6.3 op pagina 109 toont hoe dit ondersteunt werd in de code. Een action tag in de XML beschrijving bevatte een subboom (in XML) die geïnterpreteerd kon worden door een "action plugin". De voorbeelden in listings 6.5 op pagina 111, 6.6 op pagina 111 en 8.1 op pagina 133 geven respectievelijk weer hoe men directe aanroepen kan doen, Python scripts kan gebruiken en gebruik kan maken van SEESCOA componenten (zie sectie B.7).

De beslissing om XML te gebruiken voor het beschrijven interfaces wordt bekrachtigd door de opgang van XML-gebaseerde gebruikersinterface beschrijvingen tijdens de laatste jaren [LALV04]. We vinden een grote diversiteit terug, en vele andere initiatieven gaan ook veel verder dan enkel het beschrijven van de presentatie en nemen ook andere modellen op in de beschrijving. Tabellen 3.1 en 3.2 op pagina's 46 en 47 en figuur 3.1 op pagina 44 positioneren de verschillende initiatieven ten opzicht van elkaar.

B.6 Layoutbeheer voor Meerdere Apparaten

Om een gebruikersinterface bruikbaar te maken op meerdere apparaten zonder voor elk apparaat de interface opnieuw te ontwerpen en te bouwen, is er nood aan een flexibel layoutbeheersysteem. Dit systeem moet toelaten om de layout van gebruikersinterface te veranderen aan de hand van de beperkingen die opgelegd worden door het doelapparaat.

De layout moet op een generieke manier kunnen gespecificeerd worden zodanig dat de interface flexibel genoeg is om zonder wijziging op verschillende apparaten en platformen gebruikt te worden, terwijl de logisch samenhang toch behouden blijft. We hebben geopteerd om de hiërarchische structuur die we in SEESCOA XML vinden uit te buiten en tussen interactoren of groepen die op hetzelfde niveau in de hiërarchie voorkomen bepaalde beperkingen op te leggen[LCC03]. We maken gebruik van spatiële constraints om dit te verwezenlijken. Figuur 7.1 op pagina 119 toont grafisch hoe de layout op deze manier gespecificeerd kan worden. Listing 7.1 op pagina 120 toont de layout constraints beschreven in XML. Tussen de siblings in de boom kunnen layout constraints voorkomen, maar echter niet tussen verschillende niveaus in de boom.

Binnen een groep gelden de volgende regels:

- Een groep beschrijft een verzameling van logisch samenhangende interactoren.
- Een groep kan als opsplitsbaar gekenmerkt worden: hierdoor kan de layout manager beslissen de verschillende kinderen in de groep op te splitsen en op verschillende schermen te tonen.
- Een groep kan als niet opsplitsbaar gekenmerkt worden: hierdoor zal de layout manager ervoor zorgen dat deze groep altijd als één geheel getoond zal worden.

Om de layout te berekenen wordt er gebruik gemaakt van een simpel constraint oplossings-algoritme, geïnspireerd door [SMFBB93]. Dit gebeurt door een geschikte plaats te vinden in een voorgedefinieerde grid waarin de verschillende interactoren moeten gelegd worden. De dimensie van de grid wordt vooraf bepaald aan de hand van de beschikbare ruimte en de gewenste grootte of het gewicht (belang) van de interactoren. De layout zal pas berekend worden als de gebruikersinterface gerendered wordt op het doelapparaat waarbij het de beschikbare schermgrootte dan in rekening neemt. Indien de schermruimte te beperkt is om heel de interface te tonen, wordt er gebruik gemaakt van het

opsplitsen van groepen die als opsplitsbaar gekenmerkt werden. De kinderen van een splitsbare groep kunnen achter elkaar gezet worden, met behulp van tabbladen bijvoorbeeld.

B.7 Componenten en Gebruikersinterfaces voor Meerdere Apparaten

Als mogelijk applicatiemodel hebben we Dygimes gebruikt als raamwerk voor gebruikersinterfaces bovenop het SEESCOA componentensysteem. Het SEESCOA componentensysteem is een resultaat van het IWT/STWW SEESCOA project, en is een asynchroon componentensysteem voor ingebedde systemen. Het gebruik van SEESCOA componenten is tevens locatie-transparant wat zeer nuttig blijkt om migreerbare gebruikersinterfaces mee te bouwen. Vanuit het standpunt van de ontwikkelaar van de gebruikersinterface onderscheiden we 3 types van SEESCOA componenten:

Interne componenten : implementeren functionaliteit die nooit rechtstreeks in een gebruikersinterface getoond zal worden

Oppervlakte componenten : worden expliciet gemaakt in de gebruikersinterface; ze bevatten functionaliteit die gevisualiseerd dient te worden en/of waarmee de gebruiker interactie kan hebben.

Rendering componenten : hebben als functie de gebruikersinterfaces voor oppervlakte componenten te renderen op een specifiek apparaat.

Zowel interne componenten als oppervlakte componenten kunnen typisch een hoog-niveau XML-beschrijving van de gebruikersinterface bevatten die ze willen aanbieden. Het zijn echter de oppervlakte componenten die de gebruikersinterfaces van interne componenten ophalen en een rendering component aanspreken om voor de visualisatie te zorgen.

Software die gebouwd wordt door middel van SEESCOA componenten moet op deze manier geen aparte gebruikersinterface meer voorzien: de complete interface is simpelweg een aggregatie van de interfaces die alle oppervlakte componenten aanleveren. Aangezien niet alle oppervlakte componenten altijd nodig zijn, kan men een selectie maken van de oppervlakte componenten die op een bepaald moment moeten gevisualiseerd worden aan de hand van de taak die moet uitgevoerd worden (gespecificeerd in taakmodel).

Het SEESCOA componentensysteem is een mooi voorbeeld van hoe een willekeurig applicatiemodel kan gebruikt worden om de gebruikersinterface

van de juiste functionaliteit te voorzien. Daarnaast heeft de integratie met het SEESCOA componentensysteem tevens bewezen dat de UiBuilder renderer geschikt was voor ingebedde systemen.

B.8 Uiml.net: een Open Uiml Renderer voor het .Net Raamwerk

Ter vergelijking en als alternatief voor het door ons zelf ontwikkelde SEESCOA XML (ondersteund door een Java-gebaseerde renderer), hebben we een renderer voor de User Interface Markup Language ([AH04b, Pha00, APB+99, AH04a], UIML) gebouwd met behulp van C# op het .Net raamwerk. Onze renderer, Uiml.net [LC04b], is de eerste vrije renderer die de UIML 3.1 specificatie implementeert. Een UIML beschrijving van een gebruikersinterface bestaat ruwweg uit 5 delen: enerzijds de interface die 4 subdelen bevat en anderzijds de peers. De interface beschrijving maakt een strikte scheiding tussen structuur, stijl, inhoud en gedrag. De peers bevatten informatie over hoe de structuur en stijl in een concrete interface omgezet kunnen worden en hoe het gedrag kan gekoppeld worden aan echte applicaties.

Een belangrijke motivatie voor de creatie van een UIML renderer was om te breken met de voorgedefinieerde set van abstracties die beperkend werkten met SEESCOA XML. Uiml.net voorziet daarom een *reflectieve* rendering kern. De renderer heeft geen interne informatie over de widgets die gebruikt worden om de abstracties uit een UIML document om te zetten in een concrete gebruikersinterface, maar maakt gebruik van een externe beschrijving van de beschikbare mappings. Dit heeft tot gevolg dat Uiml.net geschikt is voor het gebruik van meerdere widget sets; momenteel worden Gtk#, System.Windows.Forms en Wx.NET ondersteund. Daarnaast kan Uiml.net met Mono, het Microsoft .Net raamwerk en het Microsoft .Net Compact raamwerk gebruikt worden. Dit maakt de software zelf grotendeels platform- en apparaat-onafhankelijk.

We kunnen aantonen dat UIML en SEESCOA XML perfect uitwisselbaar zijn als alternatieven voor het presentatiemodel in het Dygimes proces. SEESCOA XML en UIML bieden beiden een interface beschrijvingstaal aan die tevens kan geïntegreerd worden met een applicatiemodel en die kan gebruikt kan worden vanuit het taakmodel. Zo laat tabel 9.1 op pagina 162 zien hoe UIML als een gebruikersinterface bouwblok met een taak kan gerelateerd worden, waarna exact dezelfde procedure kan gebruikt worden om tot een werkende interface te komen. Een van de grote verschillen is de manier van layout

management: in UIML dient dit ingebakken te worden in de interface beschrijving en is grotendeels afhankelijk van de gebruikte widget set waarmee zal gerendered worden. Om dit op te vangen kan er gebruik gemaakt worden van container templates (zie tabel 9.2 op 166) om verschillende bouwblokken in één dialoog te zetten.

B.9 Context in de Ontwikkeling van Gebruikersinterfaces

Naar de toekomst toe zullen context-gevoelige gebruikersinterfaces sterk aan belang winnen. Context-gevoelige gebruikersinterfaces kunnen zich aanpassen aan de context waarin ze gebruikt worden, waarbij context ruim geïnterpreteerd wordt als aan de hand van de definitie gegeven door Dey [DSA01]: "Context is de informatie afkomstig uit de omgeving die de taken die gebruiker wil, moet of zou kunnen uitvoeren beïnvloeden".

We breiden de Dygimes omgeving uit om context in de verschillende modellen in rekening te brengen. Deze uitbreiding, Dynamo-AID [CLC04c, CLC04b], integreert context in het taakmodel door middel van beslissingsknopen. Figuren 10.2 op pagina 177 en 10.3 op pagina 178 tonen een taakmodel waarin beslissingsknopen gebruikt worden. Een beslissingsknoop verbindt een aantal alternatieve en wederzijds exclusieve subtaken en bevat een verzameling regels die aan de hand van de beschikbare context-informatie beslissen welke subtaak voor een bepaalde context in de boom geïntegreerd dient te worden. Deze beslissingsknopen vervangen zichzelf dus door één van hun subtaken gekozen aan de hand van de selectieregels die in zo een beslissingsknoop zitten. Listing 10.2 op pagina 174 toont een voorbeeld van zulke selectieregels, het schema dat de syntax van de regels bepaalt is afgebeeld in listing 10.1 op pagina 174.

Het integreren van context heeft tevens invloed op de dialoogspecificatie daar Dygimes deze genereert uit de taakspecificatie [CLC04b]. Dit kan tot onverwachte en/of ongewenste resulaten leiden in de structuur van en de navigatie doorheen de gebruikersinterface. Er wordt daarom aan een interactieve applicatie gewerkt om tijdens het design reeds werkende prototypes te genereren van context-gevoelig interfaces [CLC04a] zodanig dat de ontwerper dadelijk de resultaten van context-wijzigingen kan bekijken en zelf beperkingen op kan leggen over hoe context de gebruikersinterface mag wijzigen.

B.10 Besluit

We besluiten deze dissertatie met een terugkoppeling naar de uitdagingen die op het begin gedefinieerd werden. Van de vijf uitdagingen werden er de vooropgestelde drie ingevuld: taak-gebaseerde interfaces, ondersteuning voor een diversiteit aan apparaten en ondersteuning voor context-gevoelige interfaces. Daarnaast werd er ook wat aandacht besteed aan het verfraaien van de interfaces. Het vooropgestelde doel werd bereikt door bestaande technieken, zoals model-gebaseerde ontwikkeling van gebruikersinterfaces, te combineren met nieuwe technieken, zoals XML-gebaseerde gebruikersinterface beschrijvingen en flexibele layout management. Vele van de voorgestelde technieken kunnen apart gebruikt worden zonder dat het hele Dygimes proces moet doorlopen worden: zo kan men bijvoorbeeld de UiBuilder of Uiml.net renderers losstaand van het taak- en dialoogmodel gebruiken en kan de TaskLib applicatie gebruikt worden om apart taakspecificaties mee te verwerken.

Multi-apparaat interfaces zijn echter nog maar het topje van de berg. We hebben al een uitbreiding gedaan naar context-gevoelige interfaces, maar kijken tevens uit naar oplossingen voor het ontwerp, de implementatie en de ondersteuning van gedistribueerde interfaces (een gebruikersinterface die gelijktijdig van verschillende communicerende apparaten gebruik kan maken). Tijdens het implementeren van de verschillende technieken werd het ook duidelijk dat de huidige widget sets onvoldoende voorbereid zijn op de uitdagingen van de volgende generatie interfaces die in pervasive en ubiquitous omgevingen de communicatie met de gebruiker zullen verzorgen. De ondersteuning van de ontwikkeling gericht op context-gevoelige interfaces is al een eerste stap om deze volgende uitdagingen aan te pakken.

Bibliography

[AH04a] Marc Abrams and Jim Helms. Retrospective on UI Description Languages, Based on 7 years Experience with the User Interface Markup Language (UIML). In Luyten et al. [LALV04], pages 1–8.

[AH04b] Marc Abrams and Jim Helms. User Interface Markup Language (UIML) Specification version 3.1. Technical report, Harmonia, 2004.

[All84] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.

[APB+99] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An Appliance-Independent XML User Interface Language. *WWW8 / Computer Networks*, 31(11-16):1695–1708, 1999.

[APQA04] Mir Farooq Ali, Manuel A. Pérez-Quiñones, and Marc Abrams. *Building Multi-Platform User Interfaces with UIML*, pages 95–116. In Seffah and Javahery [SJ04], 2004.

[Bau96] Bernard Bauer. Generating User Interfaces from Formal Specifications of the Application. In Vanderdonckt [Van96], pages 141–157.

[BC95] K. Bharat and L. Cardelli. Migratory applications. In *Eighth ACM Symposium on User Interface Software and Technology*, pages 133–42, 1995.

[BCPS04] Silvia Berti, Francesco Correanim, Fabio Paternò, and Carmen Santoro. The TERESA XML Language for the Description of Interactive Systems at MultipleAbstraction Levels. In Luyten et al. [LALV04], pages 103–110.

[BHL+95] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, Isabelle Provot, Benoit Sacré, and Jean Vanderdonckt. Towards a systematic building of software architecture: The TRIDENT methodological guide. In Philippe Palanque and Rémi Bastide, editors, *Design, Specification and Verification of Interactive Systems '95*, pages 262–278, Wien, 1995. Springer-Verlag.

[BHLV94] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, and Jean Vanderdonckt. Towards a Dynamic Strategy for Computer-Aided Visual Placement. In *Workshop on Advanced Visual Interfaces*, pages 78–87. ACM press, 1994.

[BMS04] Steffen Bleul, Wolfgang Mueller, and Robbie Schaefer. Multimodel Dialog Description for Mobile Devices. In Luyten et al. [LALV04], pages 95–102.

[BMSX97] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving Lineair Arithmetic Constraints for User Interface Applications. In *Proceedings of the 13th Annual Symposium on User Interface Software and Technology (UIST-97)*, 1997.

[Bor79] Alan Borning. ThingLab – A Constraint-Oriented Simulation Laboratory. Technical report, XEROX PARC, 1979. report SSL-79-3.

[BP99] Rémi Bastide and Philippe Palanque. A Visual and Formal Glue Between Application and Interaction. *Visual Language and Computing*, 10(3), 1999.

[BS02] Carsten Binnig and Andreas Schmidt. Development of a UIML Renderer for Different Target Languages: Experiences and Design Decisions. In Kolski and Vanderdonckt [KV02], pages 267–274.

[CC03] Tim Clerckx and Karin Coninx. Integrating Task Models in Automatic User Interface Design. Technical Report TR-LUC-EDM-0302, EDM/LUC, 2003.

[CCT00] Gaëlle Calvary, Joëlle Coutaz, and David Thevenin. Embedding Plasticity in the Development Process of Interactive Systems. In *6th ERCIM Workshop "User Interfaces for All". Also in HUC (Handheld and Ubiquitous Computing) First workshop on Resource Sensitive Mobile HCI, Conference on Handheld and Ubiquitous Computing, HU2K, Bristol*, 2000.

[CCT01] Gaëlle Calvary, Joëlle Coutaz, and David Thevenin. Supporting Context Changes for Plastic User Interfaces: A Process and a Mechanism. In *Proceedings of IHM-HCI, 10-14 september 2001, Lille, France*, 2001.

[CCT+02] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Nathalie Souchon, Laurent Bouillon, and Jean Vanderdonckt. Plasticity of User Interfaces: A Revised Reference Framework. In *First International Workshop on Task Models and Diagrams for User Interface Design TAMODIA2002*, pages 127–134, July 18–19 2002.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 1999.

[CH03] Simon Crowle and Linda Hole. ISML: An Interface Specification Meta-Language. In Jorge et al. [JNF03], pages 381–396.

[CLC04a] Tim Clerckx, Kris Luyten, and Karin Coninx. Designing Interactive Systems in Context: From Prototype to Deployment. 2004. Submitted for Percom'2005.

[CLC04b] Tim Clerckx, Kris Luyten, and Karin Coninx. DynaMo-AID: a Design Process and a Runtime Architecture for Dynamic Model-Based User Interface Development. In *The 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with the 11th International Workshop on Design, Specification and Verification of Interactive Systems, Tremsbttel Castle, Hamburg, Germany*, pages 142–160, 2004.

[CLC04c] Tim Clerckx, Kris Luyten, and Karin Coninx. Generating Context-Sensitive Multiple Device Interfaces from Design. In Limbourg et al. [LJV04].

[CLC04d] Tim Clerckx, Kris Luyten, and Karin Coninx. The Mapping Problem applied to Model-Based User Interface Development for Context-Aware Applications. 2004. Submitted for Tamodia'2004.

[CLV+03] Karin Coninx, Kris Luyten, Chris Vandervelpen, Jan Van den Bergh, and Bert Creemers. Dygimes: Dynamically Generating Interfaces for Mobile Computing Devices and Embedded Systems. In Luca Chittaro, editor, *Mobile HCI*, volume 2795 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2003.

[CMP04] Francesco Correani, Guilio Mori, and Fabio Paternò. Supporting Flexible Development of Multi-Device Interfaces. In *The 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with the 11th International Workshop on Design, Specification and Verification of Interactive Systems, Tremsbttel Castle, Hamburg, Germany*, pages 161–176, 2004.

[Coc87] Gilbert Cockton. Interaction Ergonomics, Control and Separation: Open Problems in User Interface Management. *Information and Software Technology*, 29(4):176–191, 1987.

[com01] Software engineering for embedded systems using a component oriented approach; deliverable 2.2.a/3.3.a: Component composition, seescoa confidential. Technical report, Katholieke Universiteit Leuven, Vrije Universiteit Brussel, 2001.

[con00] World Wide Web consortium. *Simple Object Access Protocol (SOAP)*. World Wide Web, `http://www.w3.org/TR/SOAP/`, 2000.

[con01a] World Wide Web consortium. *Cascading Style Sheets (CSS)*. World Wide Web, `http://www.w3.org/Style/CSS/`, 2001.

[con01b] World Wide Web consortium. *Document Object Model (DOM)*. World Wide Web, `http://www.w3.org/DOM/`, 2001.

[con01c] World Wide Web consortium. *eXtensible HyperText Markup Language (XHTML)*. World Wide Web, `http://www.w3.org/MarkUp/`, 2001.

[con01d] World Wide Web consortium. *Voice eXtensible Markup Language*. World Wide Web, `http://www.w3.org/TR/voicexml/`, 2001.

[con01e] World Wide Web consortium. *Web Services Description Language specification*. World Wide Web Consortium, `http://www.w3.org/TR/wsdl`, 2001.

[con01f] World Wide Web consortium. *XForms*. World Wide Web, `http://www.w3.org/TR/xforms/`, 2001.

[con03] World Wide Web consortium. *CC/PP W3C workgroup homepage*. World Wide Web, `http://www.w3.org/Mobile/CCPP/`, 2003.

[Cou93] Joëlle Coutaz. Encyclopedia of software engineering. Wiley and sons, 1993.

[Cov01] Robin Cover. *WAP Wireless Markup Language Specification (WML)*. World Wide Web, `http://www.oasis-open.org/cover/wap-wml.html`, 2001.

[Cro04] Simon Crowle. Into the mangle: Software engineers run creases through a user interface metaphor. In Luyten et al. [LALV04], pages 47–54.

[DBS+01] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J-C. Burgelman. Scenarios For Ambient Intelligence In 2010. Technical report, IST Advisory Group – European Comission Community Research, February 2001. ftp://ftp.cordis.lu/pub/ist/docs/istagscenarios2010.pdf.

[DF04] Anke Dittmar and Peter Forbrig. The Influence of Improved Task Models on Dialogues. In Limbourg et al. [LJV04].

[DFAB04] Alan Dix, Janet Finlay, Gregory Abowd, and Russel Beale. *Human-Computer Interaction (third edition)*. Prentice Hall, 2004.

[DFR03] Anke Dittmar, Peter Forbrig, and Daniel Reichart. Model-based Development of Nomadic Applications. In *International Workshop on Mobile Computing*, 2003.

[DSA01] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction (HCI) Journal*, 16(2-4):97–166, 2001.

[ED04] David England and Min Du. *Temporal Aspects of Multi-Platform Interaction*, pages 53–68. In Seffah and Javahery [SJ04], 2004.

[EVP01] Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta. Applying Model-Based Techniques to the Development of UIs for Mobile Computers. In Sidner and Moore [SM01], pages 69–76.

[FLUV02] Peter Forbrig, Quentin Limbourg, Bodo Urban, and Jean Vanderdonckt, editors. *Interactive Systems: Design, Specification, and Verification, 9th International Workshop, DSV-IS 2002, Rostock,*

Germany, June 12–14 , Revised Papers, volume 2221 of *Lecture Notes in Computer Science*. Springer, 2002.

[FPQAS02] Mir Farooq Ali, Manuel A. Pérez-Quiõnes, Marc Abrams, and Eric Shel. Building Multi-Platform User Interfaces with UIML. In Kolski and Vanderdonckt [KV02], pages 255–266.

[FS98] Peter Forbrig and Chris Stary. From Task to Dialog: How Many and What Kind of Models do Developers Need. In Birgit Bomsdorf and Gerd Szwillus, editors, *From Task to Dialogue: Task-Based User Interface Design*, 1998. CHI'98 workshop.

[GEM94] Phil Gray, David England, and Steve McGowan. XUAN: Enhancing UAN to capture Temporal Relationships among Actions. In *Proceedings of the conference on People and computers IX*, pages 301–312. Cambridge University Press, 1994.

[Gre86] Mark Green. A Survey of Three Dialog Models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986.

[GW04] Krzysztof Gajos and Daniel S. Weld. SUPPLE: automatically generating user interfaces. In Vanderdonckt et al. [VJR04], pages 93–100.

[HC84] G. E. Hughes and M. J. Cresswell. *A Companion to Modal Logic*. Methuen, London, 1984.

[HGHW01] David Hyatt, Ben Goodger, Ian Hickson, and Chris Waterson. *XML User Interface Language (XUL) Specification 1.0*. World Wide Web, `http://www.mozilla.org/projects/xul/`, 2001.

[HPN00] Richard Han, Veronique Perret, and Mahmoud Naghshineh. Web-Splitter: a Unified XML Framework for Multi-device Collaborative Web Browsing. In *Proceedings of the 2000 ACM conference on Computer Supported Cooperative Work*, pages 221–230. ACM Press, 2000.

[HSH90] H. Rex Hartson, Antonio C. Siochi, and D. Hix. The UAN: a User-oriented Representation for Direct Manipulation Interface Designs. *ACM Transactions on Information Systems (TOS)*, 8(3):181–203, 1990.

[JNF03] Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha, editors. *Interactive Systems. Design, Specification, and Verification, 10th International Workshop, DSV-IS 2003, Funchal, Madeira Island, Portugal, June 11–13, Revised Papers*, volume 2844 of *Lecture Notes in Computer Science*. Springer, 2003.

[Joh01] Chris Johnson, editor. *Interactive Systems: Design, Specification, and Verification, 8th International Workshop, DSV-IS 2001, Glasgow, Scotland, UK, June 13–15, Revised Papers*, volume 2220 of *Lecture Notes in Computer Science*. Springer, 2001.

[JWZ93] Christian Janssen, Anetee Weisbecker, and Jürgen Ziegler. Generating User Interfaces from Data Models and Dialog Net Specifications. In *ACM Conf. on Human Aspects in Computing Systems InterCHI'93*, pages 418–423, Amsterdam, April 24–28 1993. Addison-Wesley.

[KAS96] Shiro Kawai, Hitoshi Adai, and Tadao Saito. Designing Interface Toolkit with Dynamic Selectable Modality. In *Proceedings of the second annual ACM conference on Assistive technologies*, pages 72–79, 1996.

[KV02] Christophe Kolski and Jean Vanderdonckt, editors. *Computer-Aided Design of User Interfaces III*, volume 3. Kluwer Academic, 2002.

[KWWZ04] Oskari Koskimies, Michael Wasmund, Peter Wolkerstorfer, and Thomas Ziegert. Practical Experiences with Device Independent Authoring Concepts. In Luyten et al. [LALV04], pages 17–24.

[LALV04] Kris Luyten, Marc Abrams, Quentin Limbourg, and Jean Vanderdonckt, editors. *Developing User Interfaces with XML: Advances on User Interface Description Languages*. Sattelite workshop of Advanced Visual Interfaces (AVI) 2004, Expertise Centre for Digital Media, 2004.

[LC01] Kris Luyten and Karin Coninx. An XML-based runtime user interface description language for mobile computing devices. In Johnson [Joh01], pages 17–29.

[LC04a] Kris Luyten and Karin Coninx. ImogI: Take Control Over a Context-Aware Electronic Mobile Guide for Museums. In Barbara Schmidt-Belz and Keith Cheverst, editors, *HCI in Mobile*

Guides, Glasgow, 2004. Sattelite workshop of Mobile'HCI 2004. http://research.edm.luc.ac.be/~imogi/.

[LC04b] Kris Luyten and Karin Coninx. Uiml.net: an Open Uiml Renderer for the .Net Framework. In Limbourg et al. [LJV04].

[LCC03] Kris Luyten, Bert Creemers, and Karin Coninx. Multi-device Layout Management for Mobile Computing Devices. Technical Report TR-LUC-EDM-0301, Expertise Centre for Digital Media – Limburgs Universitair Centrum, Belgium, 2003.

[LCCV03] Kris Luyten, Tim Clerckx, Karin Coninx, and Jean Vanderdonckt. Derivation of a Dialog Model for a Task Model by Activity Chain Extraction. In Jorge et al. [JNF03], pages 203–217.

[LF01] Simon Lok and Steven Feiner. A Survey of Automated Layout Techniques for Information Presentations. In *Proceedings of SmartGraphics 2001*, March 2001.

[LFHH91] L. Logrippo, M. Faci, and M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN Systems*, 23(5):325–342, 1991.

[LJV04] Quentin Limbourg, Robert Jacob, and Jean Vanderdonckt, editors. *Computer-Aided Design of User Interfaces IV*, volume 4. Kluwer Academic, 2004.

[LK93] J. Landay and T. Kaufmann. User Interface Issues in Mobile Computing. In *Fourth Workshop on Workstation Operating Systems, Napa, CA*, 1993.

[LLCR03] Kris Luyten, Tom Van Laerhoven, Karin Coninx, and Frank Van Reeth. Runtime Transformations for Modal Independent User Interface Migration. *Interacting with Computers*, 15(3):329–347, 2003.

[LS96a] Frank Lonczewski and Siegfried Schreiber. The FUSE-System: an Integrated User Interface DEsign Environment. In Vanderdonckt [Van96], pages 37–56.

[LS96b] Frank Lonczewski and Siegfried Schreiber. The FUSE-System: an Integrated User Interface Design Environment. In *Computer-Aided Design of User Interfaces*, 1996.

[LV04] Quentin Limbourg and Jean Vanderdonckt. Transformational Development of User Interfaces with Graph Transformations. In Limbourg et al. [LJV04].

[LVC02] Kris Luyten, Chris Vandervelpen, and Karin Coninx. Migratable User Interface Descriptions in Component-Based Development. In Forbrig et al. [FLUV02], pages 62–76.

[LVM+04a] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan. USIXML: A User Interface Description Language for Context-Sensitive User Interfaces. In Luyten et al. [LALV04], pages 55–62.

[LVM+04b] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Victor López-Jaquero. USIXML: a Language Supporting Multi-Path Development of User Interfaces. In *The 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with the 11th International Workshop on Design, Specification and Verification of Interactive Systems, Tremsbttel Castle, Hamburg, Germany*, pages 89–107, 2004.

[LVS00] Quentin Limbourg, Jean Vanderdonckt, and Nathalie Souchon. The Task-Dialog and Task-Presentation Mapping Problem: Some Preliminary Results. In Palanque and Paternò [PP00], pages 227–246.

[MBFB89] J. Maloney, A. Boming, and B.N. Freeman-Benson. Constraint Technology for User Interface Construction in ThingLab II. In *OOPSLA*, 1989.

[Mer01] Roland A. Merrick. Device Independent User Interfaces in XML. BelCHI Kick-off meeting, 2001. `http://www.belchi.be/event.htm`.

[MF04] Guido Menkhaus and Sebastian Fischmeister. Adaptation for Device Independent Authoring. In Luyten et al. [LALV04], pages 151–158.

[MFC01] Andreas Mülller, Peter Forbrig, and Clemens Cap. Model-Based User Interface Design Using Markup Concepts. In Johnson [Joh01], pages 30–39.

[MHP00] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000.

[MPS02] Giulio Mori, Fabio Paternò, and Carmen Santoro. CTTE: support for developing and analyzing task models for interactive system design. *IEEE Transactions on Software Engineering*, 28(8):797–813, 2002.

[MPS03] Guilio Mori, Fabio Paternò, and Carmen Santoro. Tool Support for Designing Nomadic Applications. In *Proceedings of the 2003 international conference on Intelligent user interfaces*, pages 141 – 148, Miami, Florida, USA, January 12–15 2003.

[MPS04] Giulo Mori, Fabio Paternò, and Carmen Santoro. Design and development of multidevice user interface through multiple logical descriptions. *Transactions on Software Engineering*, 30(8), August 2004.

[MT02] Ian Main and The GTK Team. *GTK+ 2.0 Tutorial*. World Wide Web, `http://www.gtk.org/tutorial`, 2002.

[MV02] Efrem Mbaki and Jean Vanderdonckt. Window Transitions: A Graphical Notation for Specifying Mid-level Dialogue. In *First International Workshop on Task Models and Diagrams for User Interface Design TAMODIA2002*, pages 55–63, July 18–19 2002.

[MWK04] Roland A. Merrick, Brian Wood, and William Krebs. Abstract User Interface Markup Language. In Luyten et al. [LALV04], pages 39–46.

[NMH+02] Jeffrey Nichols, Brad A. Myers, Michael Higgins, Joseph Hughes, Thomas K. Harris, Roni Rosenfeld, and Mathilde Pignol. Generating remote control interfaces for complex appliances. In *User Interface Software and Technology*, 2002.

[Ols92] Dan Olsen. *User Interface Management Systems: Models and Algorithms*. Morgan Kaufman Publishers Inc., 1992.

[Par69] David L. Parnas. On the use of transition diagrams in the design of a user interface for an interactive computer system. In *Proceedings of the 1969 24th national conference*, pages 379–385, 1969.

[Pat97] Fabio Paternò. Formal Reasoning about Dialogue Properties with Automatic Support. *Interacting with Computers*, 9(2):173–196, November3 1997.

[Pat00] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 2000.

[PE99] Angel Puerta and Jacob Eisenstein. Towards a General Computational Framework for Model-Based Interface Development Systems. In *IUI 1999 International Conference on Intelligent User Interfaces*, pages 171–178, 1999.

[PE02] Angel Puerta and Jacob Eisenstein. XiML: A Common Representation for Interaction Data. In *Sixth International Conference on Intelligent User Interfaces*, pages 214–215, 2002.

[PE04] Angel Puerta and Jacob Eisenstein. *XIML: A Multiple User Interface Representation Framework for Industry*, pages 119–148. In Seffah and Javahery [SJ04], 2004.

[Pha00] Constantinos Phanouriou. *UIML: A Device-Independent User Interface Markup Language*. PhD thesis, Virginia Tech, 2000.

[Pin00] Paulo Pinheiro da Silva. User Interface Declarative Models and Development Environments: A Survey. In Palanque and Paternò [PP00], pages 207–226.

[PL94] Fabio Paterno and Ales Leonardi. A Semantics-based Approach for the Design and Implementation of Interaction Objects. *Computer Graphics Forum*, 13(3):195–204, 1994.

[PLV01] Costin Pribeanu, Quentin Limbourg, and Jean Vanderdonckt. Task Modelling for Context-Sensitive User Interfaces. In Johnson [Joh01], pages 60–76.

[PP00] Philippe Palanque and Fabio Paternò, editors. *Interactive Systems: Design, Specification, and Verification, 7th International Workshop, DSV-IS 2000, Limerick, Ireland, June 5-6, Revised Papers*, volume 1946 of *Lecture Notes in Computer Science*. Springer, 2000.

[PS02] Fabio Paternò and Carmen Santoro. One model, many interfaces. In Kolski and Vanderdonckt [KV02], pages 143–154.

[Pue97] Angel R. Puerta. A model-based interface development environment. *IEEE Softw.*, 14(4):40–47, 1997.

[PVW+04] Davy Preuveneers, Jan Van den Bergh, Dennis Wagelaar, Andy Georges, Peter Rigole, Tim Clerckx, Yolande Berbers, Karin Coninx, Viviane Jonckers, and Koen De Bosschere. Towards an Extensible Context Ontology for Ambient Intelligence. In *Proceedings of EUSAI 2004*, November 8–10 2004. Accepted for publication.

[RB03] Peter Rigole and Yolande Berbers. The working of the SEESCOA common test case. Technical Report Report CW 354, Department of Computer Science – K.U.Leuven, Belgium, 2003.

[Sch96] Egbert Schlungbaum. Model-based User Interface Software Tools - Current State of Declarative Models. Technical Report 96-30, Graphics, Visualization and Usability Center – Georgia Institute of Technology Atlanta, 1996.

[SE96a] Egbert Schlungbaum and Thomas Elwert. Automatic User Interface Generation from Declarative Models. In Vanderdonckt [Van96], pages 3–17.

[SE96b] Egbert Schlungbaum and Thomas Elwert. Dialogue Graphs - a formal and visual specification technique for dialogue modelling. In *Formal Aspects of the Human Computer Interface*, 1996.

[SJ04] Ahmed Seffah and Homa Javahery, editors. *Multiple User Interfaces – Cross-platform Applications and Context-aware Interfaces.* Wiley, 2004.

[SLN92] Pedro A. Szekely, Ping Luo, and Robert Neches. Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design. In *CHI*, pages 507–515, 1992.

[SLV02] Nathalie Souchon, Quentin Limbourg, and Jean Vanderdonckt. Task Modelling in Multiple contexts of Use. In Forbrig et al. [FLUV02], pages 60–76.

[SM01] Candy Sidner and Johanna Moore, editors. *Proceedings of the 2001 International Conference on Intelligent User Interfaces, January 14-17, 2001 , Santa Fe, New Mexico.* ACM, 2001.

[SMFBB93] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software - Practice and Experience*, 23(5):529–566, 1993.

[Som04] Ian Sommerville. *Software Engineering*. Addison-Wesley, 7th edition, 2004.

[SR98] Kurt Stirewalt and S. Rugaber. Automating User-Interface Generation by Model Composition . In *Proceedings of the IEEE International Conference on Automated Software Engineering* , 1998.

[SR01] Christian Sandor and Thomas Reicher. CUIML: A Language for the Generation of Multimodal Human-Computer Interfaces. In *Proceedings of the European UIML Conference*, 2001.

[SSC+95] Pedro A. Szekely, Piyawadee Noi Sukaviriya, Pablo Castells, Jeyakumar Muthukumarasamy, and Ewald Salcher. Declarative Interface Models for User Interface Construction Tools: The MASTERMIND Approach. In *EHCI*, pages 120–150, 1995.

[Sti97] Kurt Stirewalt. *Automatic Generation of Interactive Systems from Declarative Models*. PhD thesis, Georgia Institute of Technology, 1997.

[Sti99] Kurt Stirewalt. MDL: a Language for Binding UI Models. In Vanderdonckt and Puerta [VP99], pages 159–170.

[SV03] Nathalie Souchon and Jean Vanderdonckt. A Review of XML-complaint User Interface Description Languages. In Jorge et al. [JNF03], pages 377–391.

[Sze96] Pedro Szekely. Retrospective and Challenges for Model-Based Interface Development. In Vanderdonckt [Van96], pages xxi–xliv.

[Szy98] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998.

[TC99] David Thevenin and Joëlle Coutaz. Adaptation and Plasticity of User Interfaces. In *Workshop on Adaptive Design of Interactive Multimedia Presentations for Mobile Users*, 1999.

[The01] David Thevenin. *Adaptation en Interaction Homme-Machine : le cas de la Plasticité*. PhD thesis, Université Joseph Fourier - Grenoble 1, 2001.

[UBB01] David Urting, Stefan Van Baelen, and Yolande Berbers. Embedded Software using Components and Contracts (Position Paper). In *European Conference for Object-Oriented Programming*, 2001.

[UBHB01] David Urting, Stefan Van Baelen, Tom Holvoet, and Yolande Berbers. Embedded Software Development: Components and Contracts. In *Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems*, pages 685–690, 2001.

[Van93] Jean Vanderdonckt, editor. *Computer-Aided Design of User Interfaces I*, volume 1. Kluwer Academic, 1993.

[Van95] Jean Vanderdonckt. Knowledge-Based Systems for Automated User Interface Generation : the TRIDENT Experience. In *Proceedings of the CHI '95 Workshop on Knowledge-Based Support for the User Interface Design Process*, May 1995.

[Van96] Jean Vanderdonckt, editor. *Computer-Aided Design of User Interfaces II*, volume 2. Kluwer Academic, 1996.

[VB93] Jean Vanderdonckt and François Bodart. Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection. In *ACM Conference on Human Aspects in Computing Systems InterCHI'93*, pages 424–429. Addison Wesley, 1993.

[VC04] Chris Vandervelpen and Karin Coninx. Towards Model-Based Design Support for Distributed User Interfaces. 2004. Accepted for publication at NordiCHI 2004, 23-27 October, Tampere, Findland.

[VG94] Jean Vanderdonckt and Xavier Gillo. Visual Techniques for Traditional and Multimedia Layouts. In *Advanced Visual Interfaces*, pages 95–104, 1994.

[VJR04] Jean Vanderdonckt, Nuno Jardim Nunes, and Charles Rich, editors. *Proceedings of the 2004 International Conference on Intelligent User Interfaces, January 13-16, 2004, Funchal, Madeira, Portugal.* ACM, 2004.

[VLC03a] Jan Van den Bergh, Kris Luyten, and Karin Coninx. A Runtime System for Context-Aware Multi-Device User Interfaces. In *HCI International 2003, Volume 2, Crete, Greece*, pages 308–312. Lawrence Erlbaum Associates, June 2003.

[VLC03b] Chris Vandervelpen, Kris Luyten, and Karin Coninx. Location-Transparent User Interaction for Heterogeneous Environments. In Constantine Stephanidis and Julie Jacko, editors, *Human-Computer Interaction: Theory and Practice (Part II), Volume 2*, pages 313–317. Lawrence Erlbaum Associate, June 2003.

[VLC04] Jan Van den Bergh, Kris Luyten, and Karin Coninx. Evaluation of High-Level user Interface Description Languages for Use on Mobile and Embedded Devices . In Luyten et al. [LALV04], pages 87–94.

[VLF03] Jean Vanderdonckt, Quentin Limbourg, and Murielle Florins. Deriving the Navigational Structure of a User Interface. In M. Rauterberg and J. Wesson, editors, *Proceedings of 9th IFIP Conf. on Human-Computer Interaction Interact'2003 (Zrich, 1-5 September 2003)*, pages 455–462, 2003.

[VP99] Jean Vanderdonckt and Angel Puerta, editors. *Computer-Aided Design of User Interfaces III*, volume 3. Kluwer Academic, 1999.

[Was85] Anthony Wasserman. Extending State Transition Diagrams for the Specification of Human-Computer Interaction. *IEEE Transactions on Software Engineering*, 11:699–713, 1985.

[WC] Kathy Walrath and Mary Campione. *The Swing Tutorial*. World Wide Web, `http://java.sun.com/docs/books/tutorial/books/swing/index.html`.

[WD90] Won Chul Kim and James D. Foley. DON: User Interface Presentation Assistant. In *User Interface Software and Technology*. ACM Press, October 1990.

[ZMBR04] Detlef Zuehlke, Kizito Mukasa, Alexander Boedcher, and Achim Reuther. useML: A Human-Machine Interface Description Language. In Luyten et al. [LALV04], pages 119–126.

Index

Model-Based User Interface Development, 9

abastract model, 70
abstract interaction objects, 26
abstract models, 19
Abstract User Interface Markup Language, 35
action, 19, 104
action plugin, 110
activity chain, 84
AIO, 26
Allen's temporal logic, 22
annotation tool, 82
application model, 11
AUIML, 35, 96
automated transformation, 70

BOSS, 16
building block annotator, 66

CADUI, 9
candidate transition, 85, 87
Cascading StyleSheets, 35
CC/PP, 34
CIO, 26
CoDAMoS, 185
common denominator, 32
component
  definition, 129
  internal, 130
  rendering, 132
  surface, 130
compound task sets, 73
concrete interaction objects, 26
concrete model, 70
concrete models, 19
ConcurTaskTree, 20
  environment, 15
  temporal operators, 21
ConcurTaskTrees, 15
Consensus, 36
constraints, 98
context, 169–183
  definition, 170
context model, 11
context-sensitive system, 186
CSS, 35, 99
CUIML, 149

data model, 11
decision nodes, 171
decision tree, 171
description language, 98
  properties, 98
design-time tool, 13
Dialog and Interface Specification Language, 40
dialog model, 11, 23
  definition, 24
  full dialog coverage, 25
  state transition network, 91
  transitions, 93
DISL, 40, 149
distributed user interfaces, 26, 187
Document Object Model, 34
DOM, 34
domain model, 11
DON, 12
DSV-IS, 9
DTD, 99
Dygimes, 12, 15, 52, 71, 72, 159, 185, 189

activity chain, 84
context-sensitive, 171
design cycle, 72
process, 72
state transition network, 91
UiBuilder, 105
UIML, 159
dynamic model, 28

enabled task collection, 22
enabled task set, 22, 77
algorithm, 77
definition, 22
enabled task sets, 74
environmental model, 171
executable model, 19
eXtended User Agent Notation, 21

full coverage, 25, 26, 161
dialog, 25
presentation, 26
full presentation coverage, 27
FUSE, 13, 16

Genius, 71
group, 104
Gtk#, 144

Humanoid, 13

ICO, 71
intent-based, 35
inter-model relationships, 27
inter-vocabulary distance, 154
interaction session, 137
Interactive Cooperative Objects, 71
Interface Specification Meta-Language, 39
intermediate model, 24
internal component, 130
IUI, 10

JavaScript, 35

layout, 95, 110
grid-based, 110
layout specification tool, 67
layout system, 187
LOTOS, 21

mapping problem, 27
Mastermind, 11
MDA, 188
meta-widget, 32
metaphor based, 39
MIM, 145
Mobi-D, 12
model
definition, 16
model derivation, 27
model drive architecture, 188
model linking, 28
model manipulation, 28
model update, 28
model-based, 10, 13
architecture, 13
derivation, 27
dialog model, 23
environment, 13, 18
environmental model, 171
Formalization, 16
linking, 28
manipulation, 28
mapping, 27
platform model, 171
presentation model, 25
relationships, 27
system, 13
task model, 19
tools, 65
update, 28
user interface development, 10
Mono, 149
Mozilla, 34
multi-device, 10

pagination, 36
Panel Definition Markup Language, 35
Pattern User Action Notation, 22
PDA, 39
plasticity, 28, 187
dialog plasticity, 28
platform model, 171
presenation model, 11
presentation model, 25, 36, 95, 97
aspects, 36, 95
definition, 26
full presentation coverage, 27

layout, 95
rendering hints, 96
requirements, 97
structure, 95
widget mappings, 96
priority order, 80
priority tree, 80
PUAN, 22
Python, 110, 111

rational unified process, 188
RDF, 35, 99
reflection, 150, 164
renderer, 105
Renderer Independent Markup Language, 36
rendering component, 132
rendering engine, 185
rendering hints, 96
Resource Description Framework, 35
ROOM, 129
run-time system, 13
run-time tool, 13
RUP, 188

SEESCOA, 7, 96, 127
notation, 130
SEESCOA XML, 7, 96, 100, 164
action, 104, 110
DTD, 101
event handling, 110
group, 104
interactors, 101
renderer, 105
versus UIML, 164
semi-dynamic model, 28
software components, 127
state transition diagram, 70
state transition network, 70, 91
static model, 28
structure, 95
surface component, 130
surveillance camera, 103
SWF, 144
System.Windows.Forms, 144

Tadeus, 12, 71
task analysis, 19
task model, 11, 19
building block annotator, 66
ConcurTaskTree, 20
definition, 19
TaskLib, 67
template, 163
Teresa, 15, 38, 71
TIDE, 144
tools, 65
Trident, 12

UAN, 21
UAprofile, 34
UiBuilder, 66, 105
class diagram, 109
extensibility, 108
UIML, 7, 40, 96, 108, 113, 143–165
architecture, 150
behavior, 145
container, 163
content, 145
full coverage, 161
layout, 157
meta-interface model, 145
structure, 145
style, 145
template, 163
versus SEESCOA XML, 164
Uiml.net, 7, 150
UML, 188
useML, 149
user interface building block, 66, 161
user interface description language
requirements, 97
User Interface Markup Language, 40
user model, 11
Useware Markup Language, 37
UsiXML, 42

vocabulary generator, 150

widget mapping, 96
windows transitions, 71
Wx.NET, 144

Xaml, 34
XForms, 34, 36, 96, 101, 113
XHTML, 36
XIML, 41, 96, 113

XML User interface Language, 34
XSLT, 99
XUAN, 21
XUL, 34, 96, 113